Program Design

Invasion Percolation: Tuning

Our program works correctly

Our program works correctly

At least, it passes all our tests

Our program works correctly

At least, it passes all our tests

Next step: figure out if we need to make it faster

Our program works correctly

At least, it passes all our tests

Next step: figure out if we need to make it faster

Spend time on other things if it's fast enough

How do we measure a program's speed?

How do we measure a program's speed?

Average of many running times on various grids

How do we measure a program's speed?

Average of many running times on various grids

How do we *predict* running time on bigger grids?

How do we measure a program's speed?

Average of many running times on various grids

How do we *predict* running time on bigger grids?

Use *asymptotic analysis*

How do we measure a program's speed?

Average of many running times on various grids

How do we *predict* running time on bigger grids?

Use *asymptotic analysis*

One of the most powerful theoretical tools in computer science

5	3	7	2	6	1	1	3	4
8	5	6	5	7	2	3	6	2
2	5	8	7	5	5	6	5	9
5	2	6	4	9	3	9	6	5
4	6	8	8	5	9	7	3	9
7	6	4	5	1	2	6	8	5
5	4	2	5	8	5	5	5	8
5	7	5	1	5	3	8	5	5
4	5	1	9	7	8	6	5	1

N

N

$N \times N$ grid has

N^2 cells

5	3	7	2	−1	1	1	3	4
8	5	6	5	−1	2	3	6	2
2	5	8	7	−1	5	6	5	9
5	2	6	4	−1	3	9	6	5
4	6	8	8	−1	9	7	3	9
7	6	4	5	1	2	6	8	5
5	4	2	5	8	5	5	5	8
5	7	5	1	5	3	8	5	5
4	5	1	9	7	8	6	5	1

N

N

$N \times N$ grid has

N^2 cells

Must fill at least N

to reach boundary

5	3	7	2	−1	1	1	3	4
8	−1	−1	−1	−1	−1	−1	−1	2
2	−1	−1	−1	−1	−1	−1	−1	9
5	−1	−1	−1	−1	−1	−1	−1	5
4	−1	−1	−1	−1	−1	−1	−1	9
7	−1	−1	−1	−1	−1	−1	−1	5
5	−1	−1	−1	−1	−1	−1	−1	8
5	−1	−1	−1	−1	−1	−1	−1	5
4	5	1	9	7	8	6	5	1

N

N

$N \times N$ grid has

N^2 cells

Must fill at least N

to reach boundary

Can fill at most

$(N-2)^2+1 = N^2-4N+5$

5	3	7	2	−1	1	1	3	4
8	−1	−1	−1	−1	−1	−1	−1	2
2	−1	−1	−1	−1	−1	−1	−1	9
5	−1	−1	−1	−1	−1	−1	−1	5
4	−1	−1	−1	−1	−1	−1	−1	9
7	−1	−1	−1	−1	−1	−1	−1	5
5	−1	−1	−1	−1	−1	−1	−1	8
5	−1	−1	−1	−1	−1	−1	−1	5
4	5	1	9	7	8	6	5	1

N

N

$N \times N$ grid has

N^2 cells

Must fill at least N

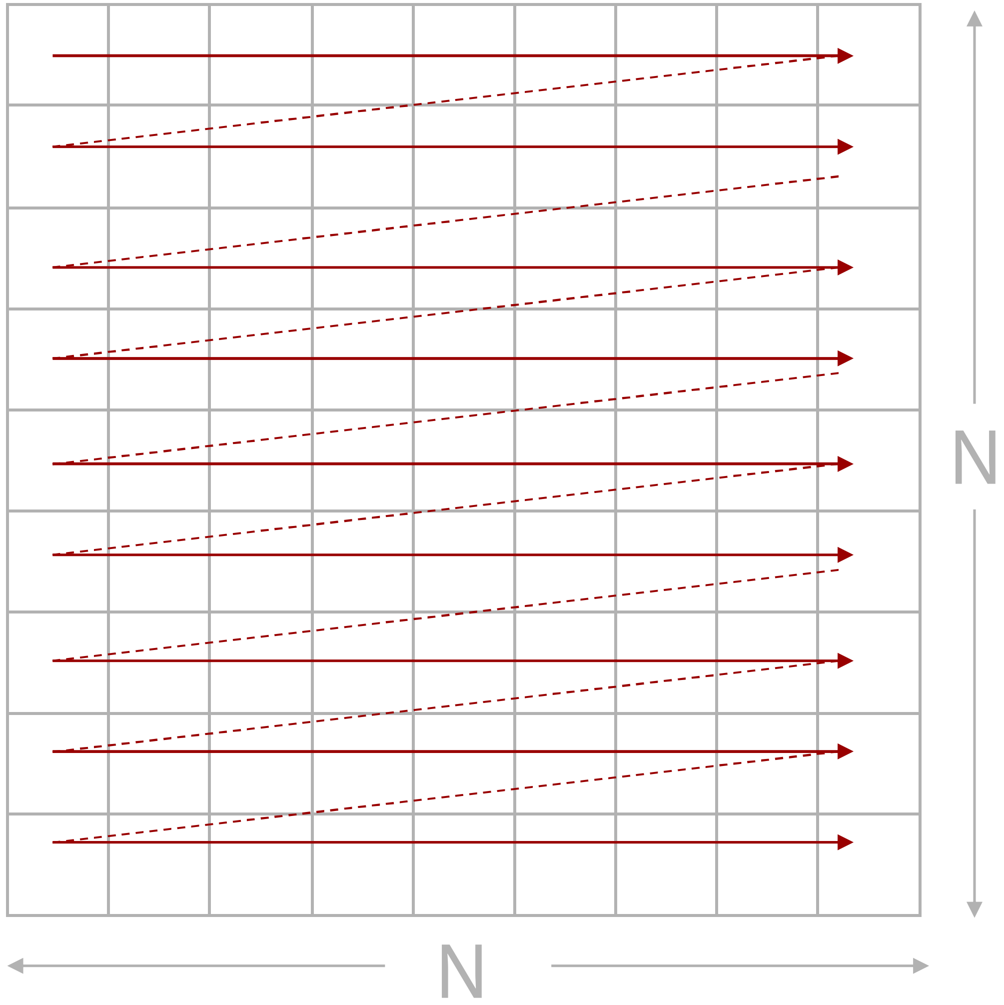
to reach boundary

Can fill at most

$(N-2)^2+1 = N^2-4N+5$

For large N, this is

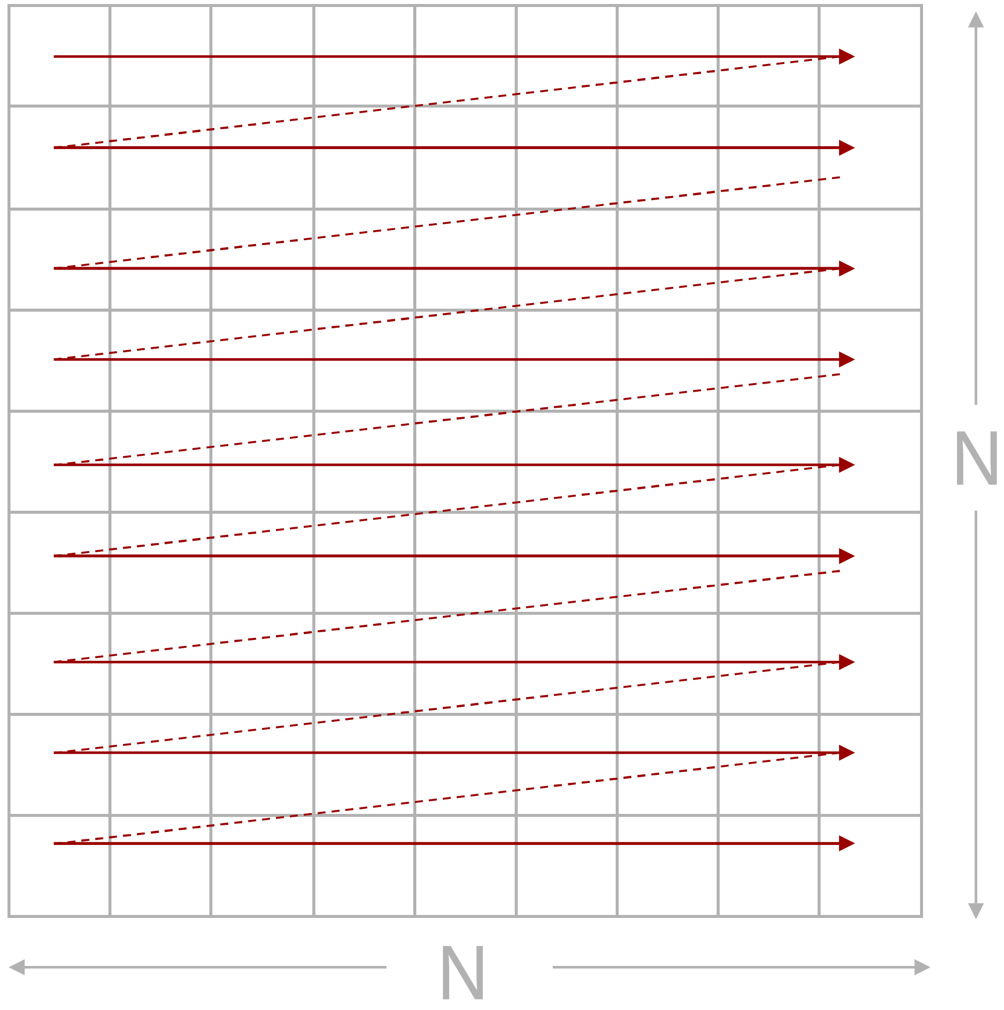
approximately N^2



Program looks at

N^2 cells to find the

next cell to fill



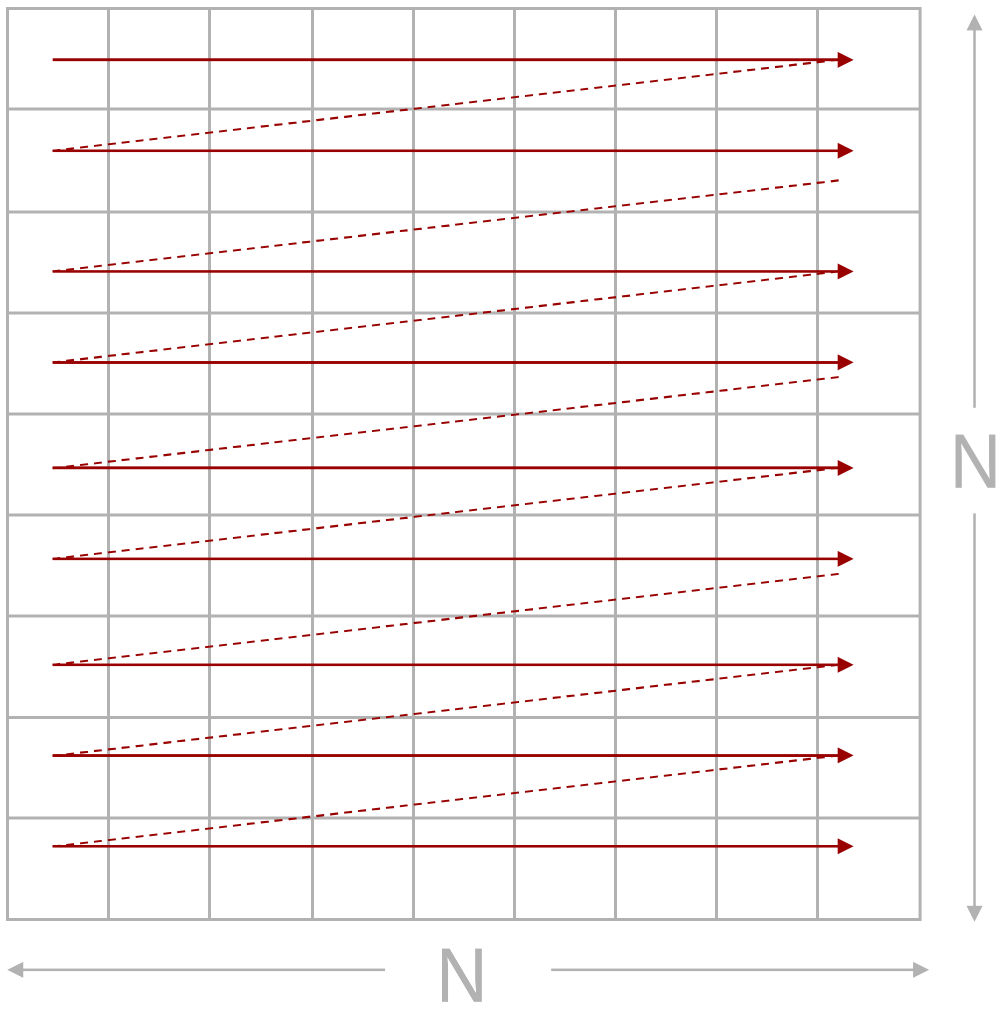
Program looks at

N^2 cells to find the

next cell to fill

Best case: $N \cdot N^2$ or

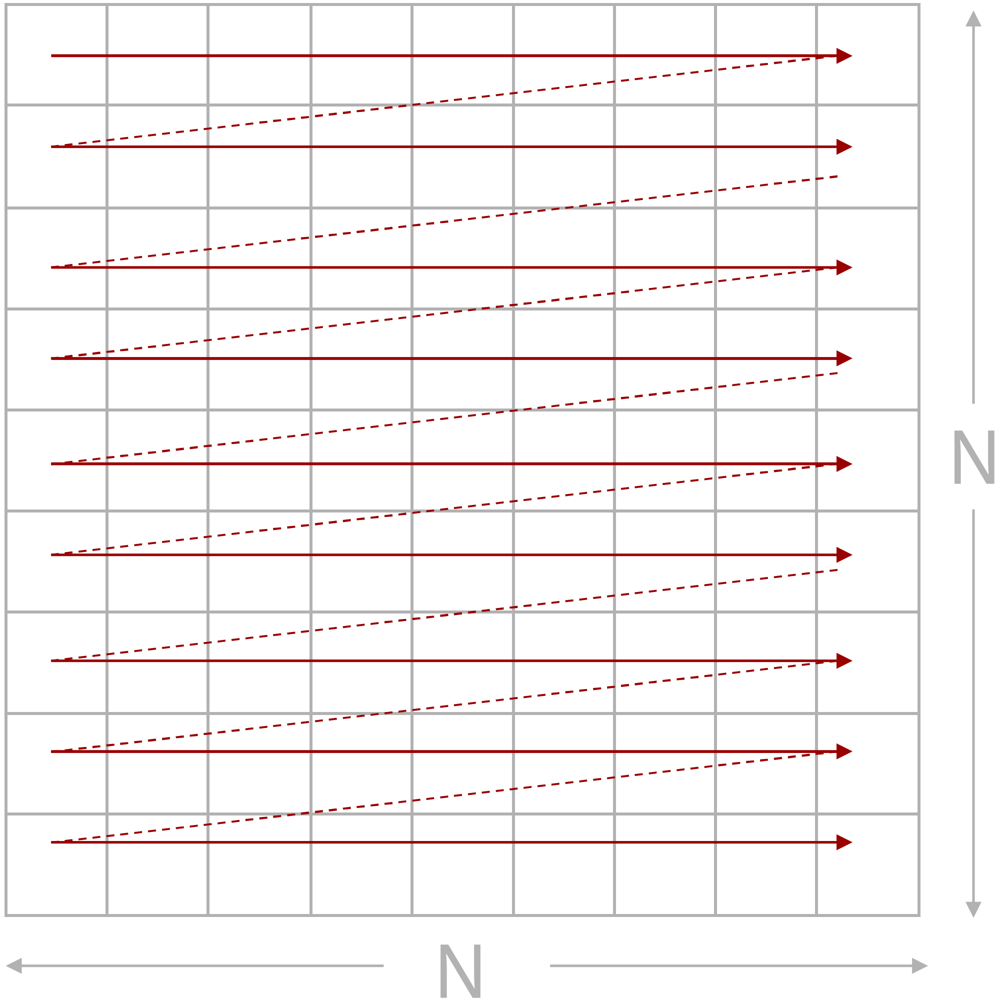
N^3 steps in total



Program looks at N^2 cells to find the next cell to fill

Best case: $N \cdot N^2$ or N^3 steps in total

Worst case: $N^2 \cdot N^2$ or N^4 steps



Program looks at

N^2 cells to find the

next cell to fill

Best case: $N \cdot N^2$

or N^3 steps in total

Worst case: $N^2 \cdot N^2$

or N^4 steps

Ouch

Averaging exponent to $N^{3.5}$ doesn't make sense...

Averaging exponent to $N^{3.5}$ doesn't make sense...

...but it will illustrate our problem

Averaging exponent to $N^{3.5}$ doesn't make sense...

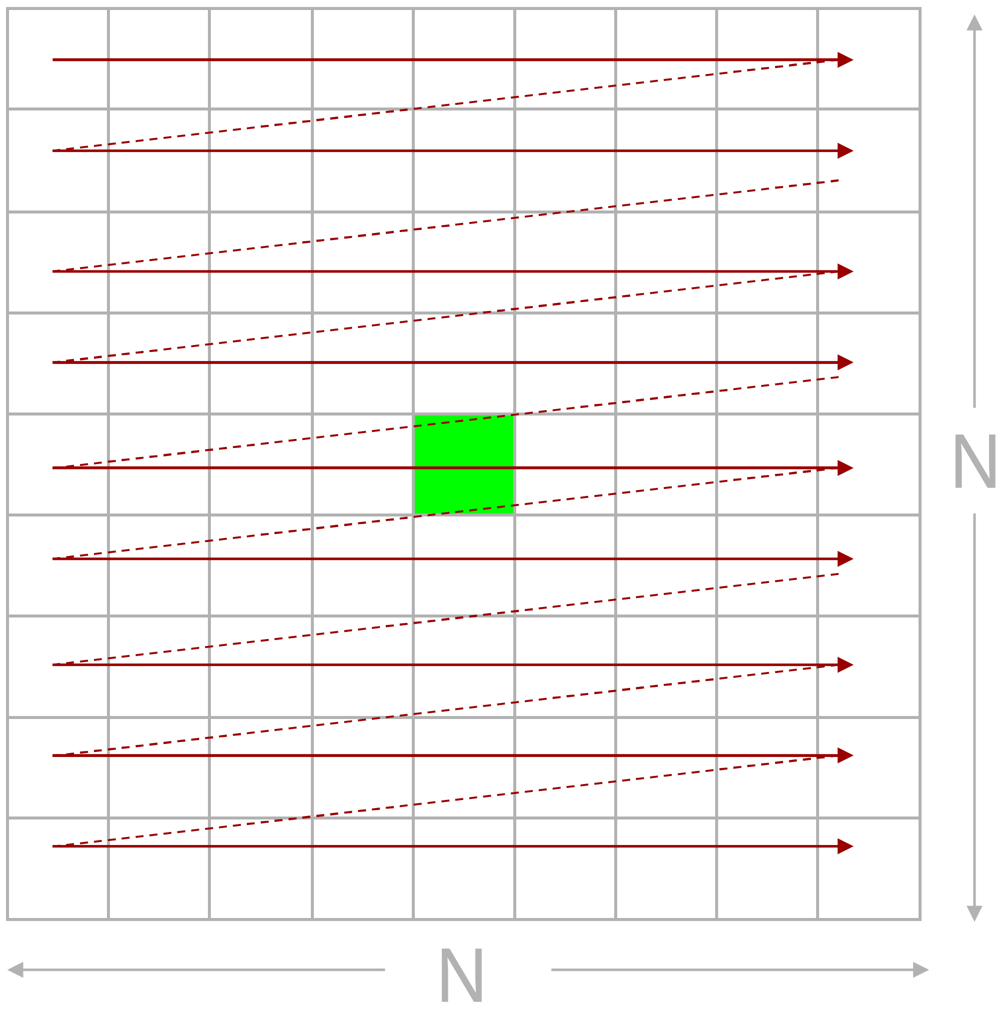
...but it will illustrate our problem

Grid Size	Time
N	T
2N	11.3 T
3N	46.7 T
4N	128 T

Averaging exponent to $N^{3.5}$ doesn't make sense...

...but it will illustrate our problem

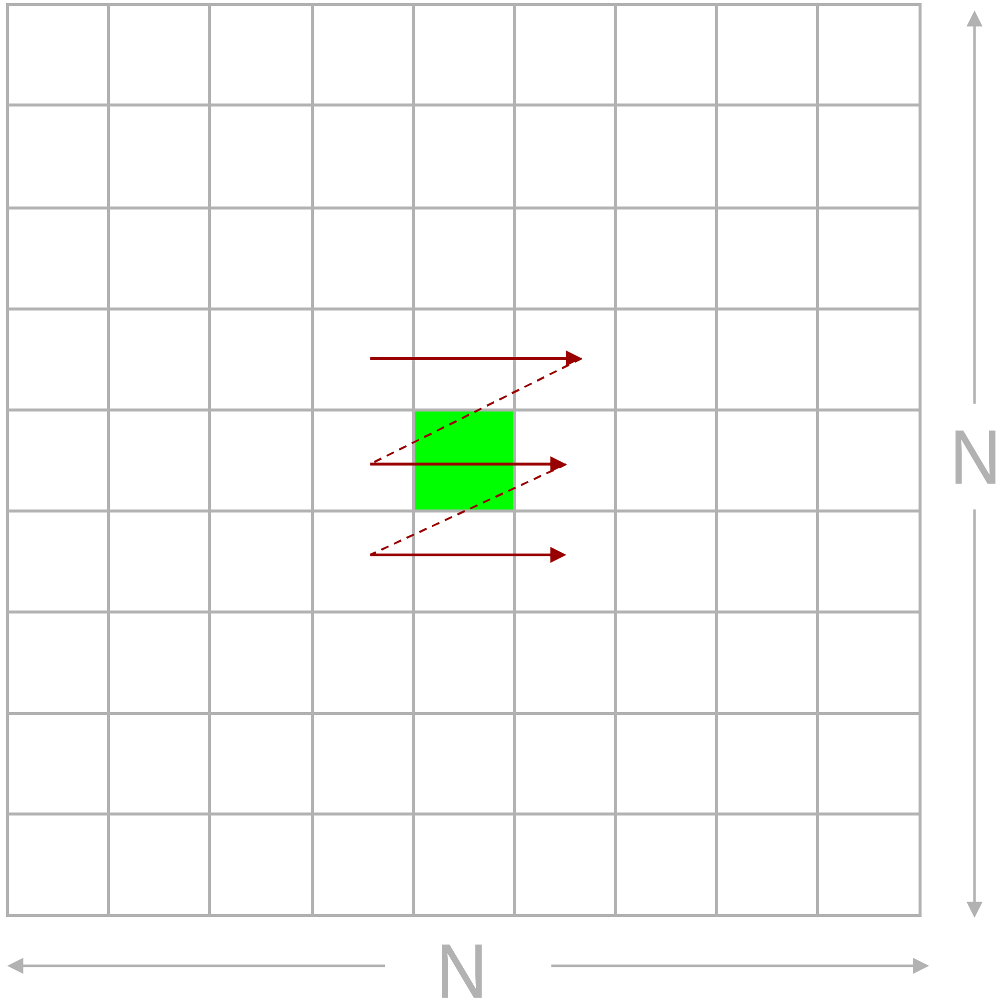
Grid Size	Time	Which Is...
N	T	1 sec
2N	11.3 T	11 sec
3N	46.7 T	47 sec
4N	128 T	2 min
10N	3162 T	52 min
50N	883883 T	1 day
100N	10^7 T	115 days



Idea #1:

Why are we looking

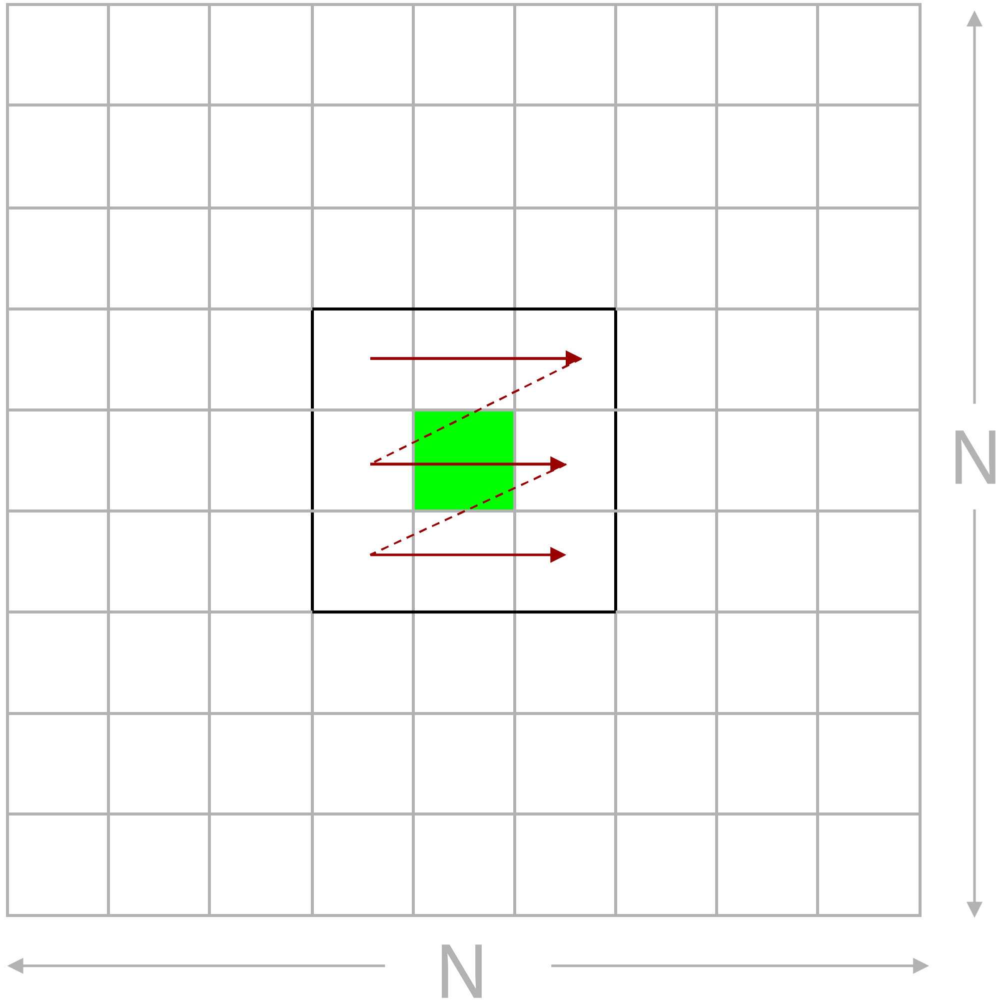
at all the cells?



Idea #1:

Why are we looking at all the cells?

Just look at cells that *might* be adjacent

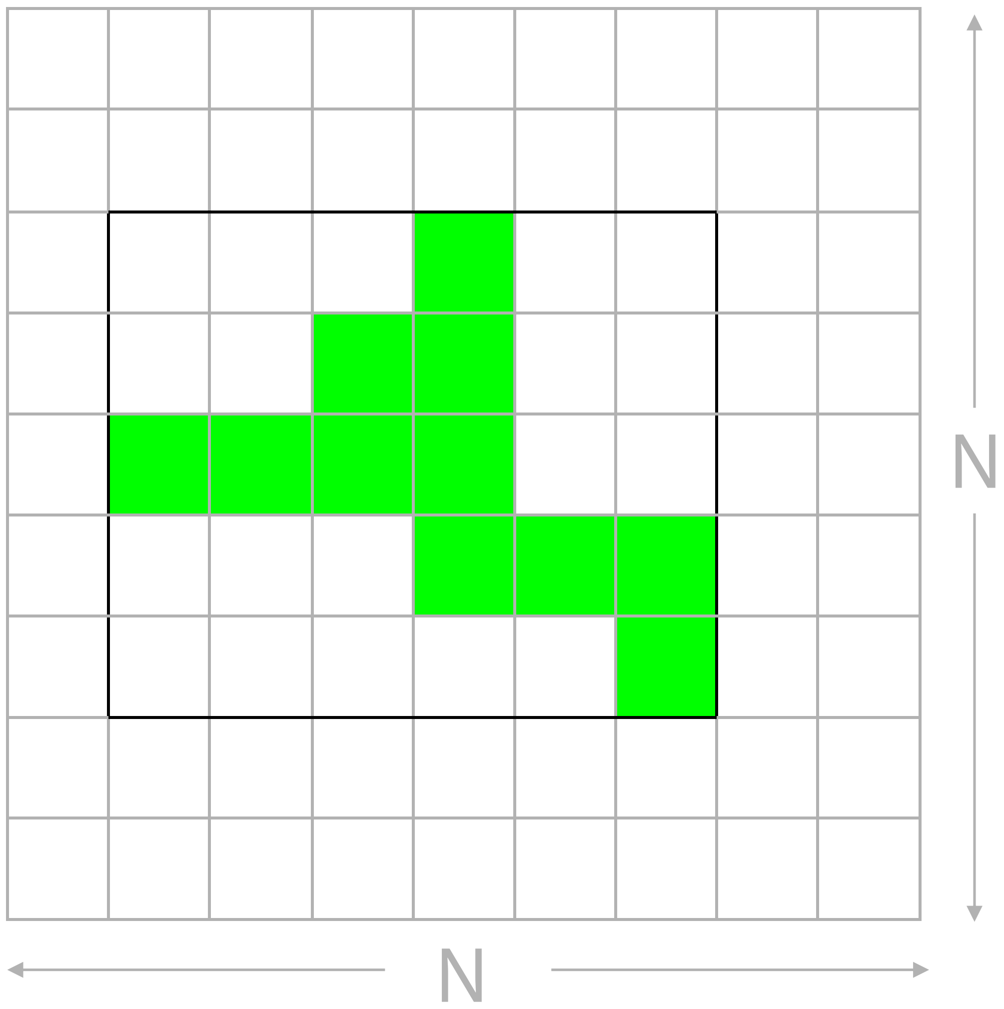


Idea #1:

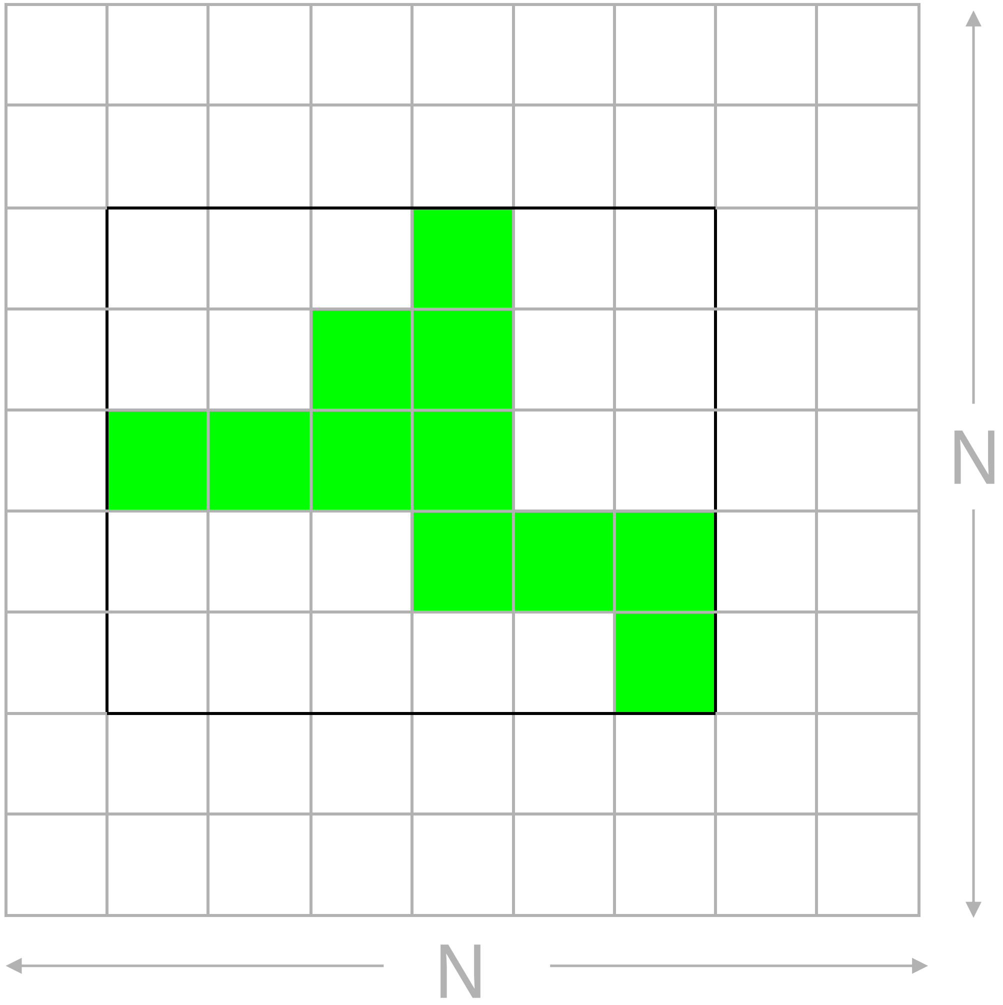
Why are we looking at all the cells?

Just look at cells that *might* be adjacent

Keep track of min and max X and Y and loop over those

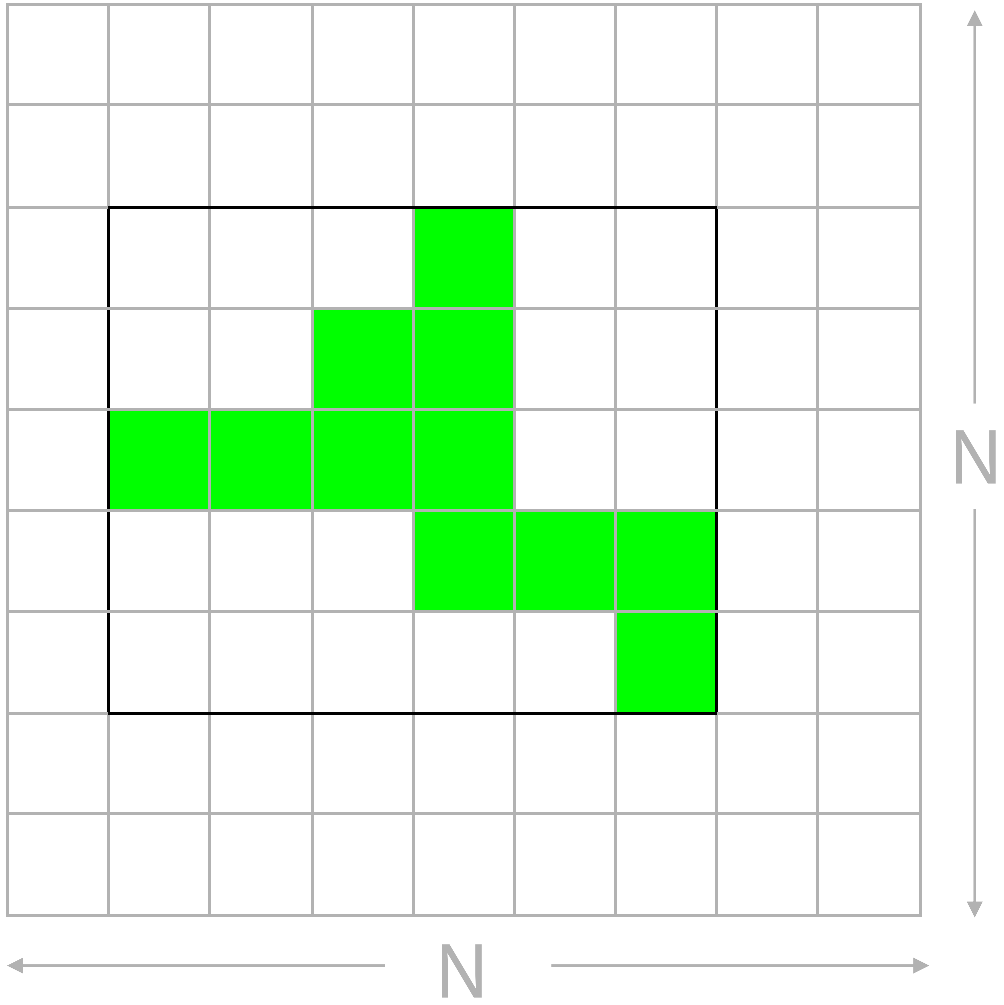


But on average, the filled region is half the size of the grid



But on average, the filled region is half the size of the grid

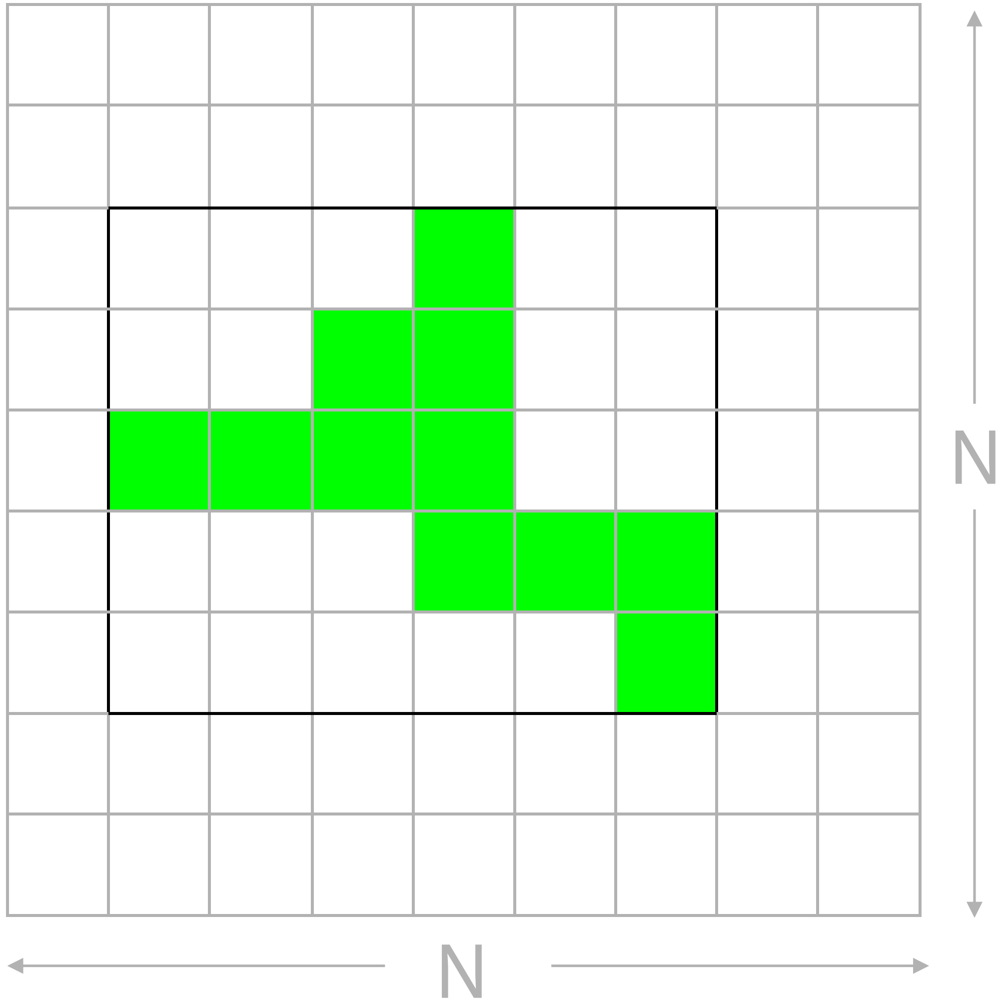
$N/2 \cdot N/2 = N^2/4$



But on average, the filled region is half the size of the grid

$N/2 \cdot N/2 = N^2/4$

115 days → 29 days

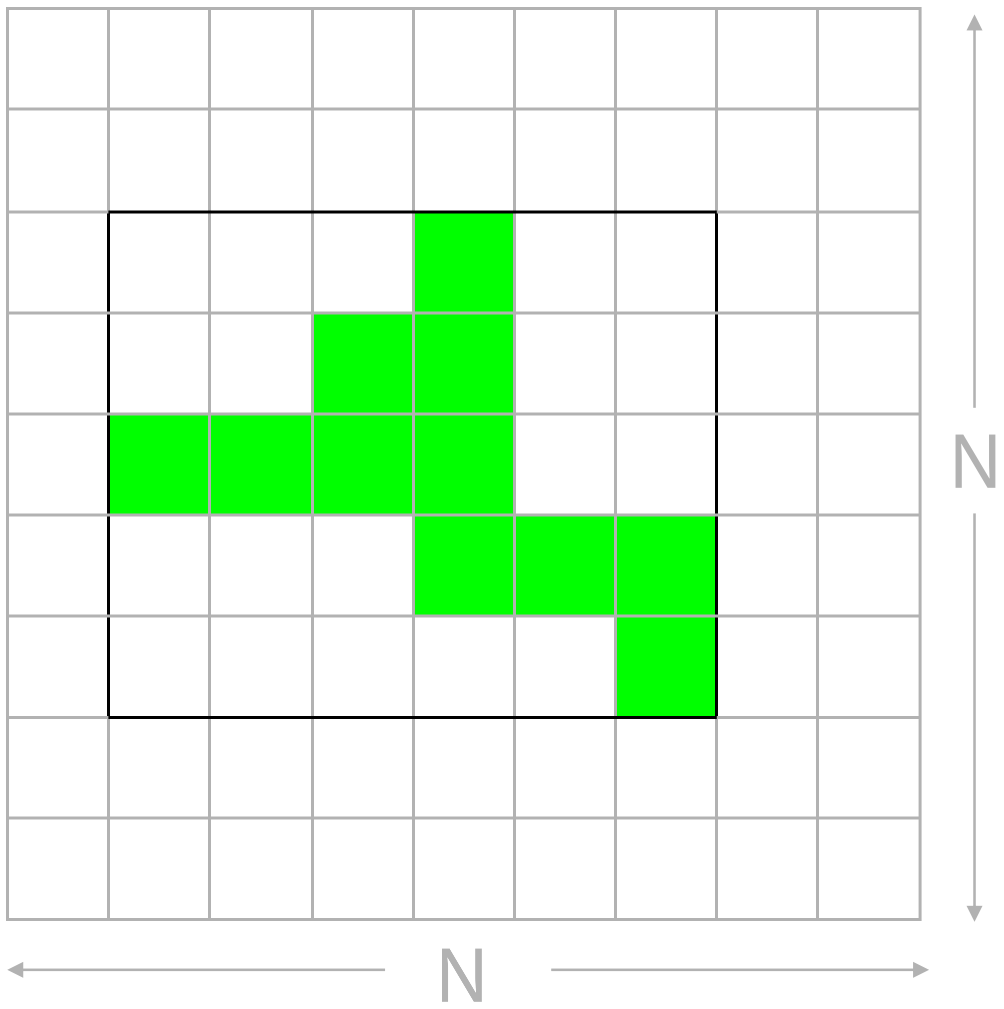


But on average, the filled region is half the size of the grid

$N/2 \cdot N/2 = N^2/4$

115 days $\rightarrow$ 29 days

148N×148N grid eats that up



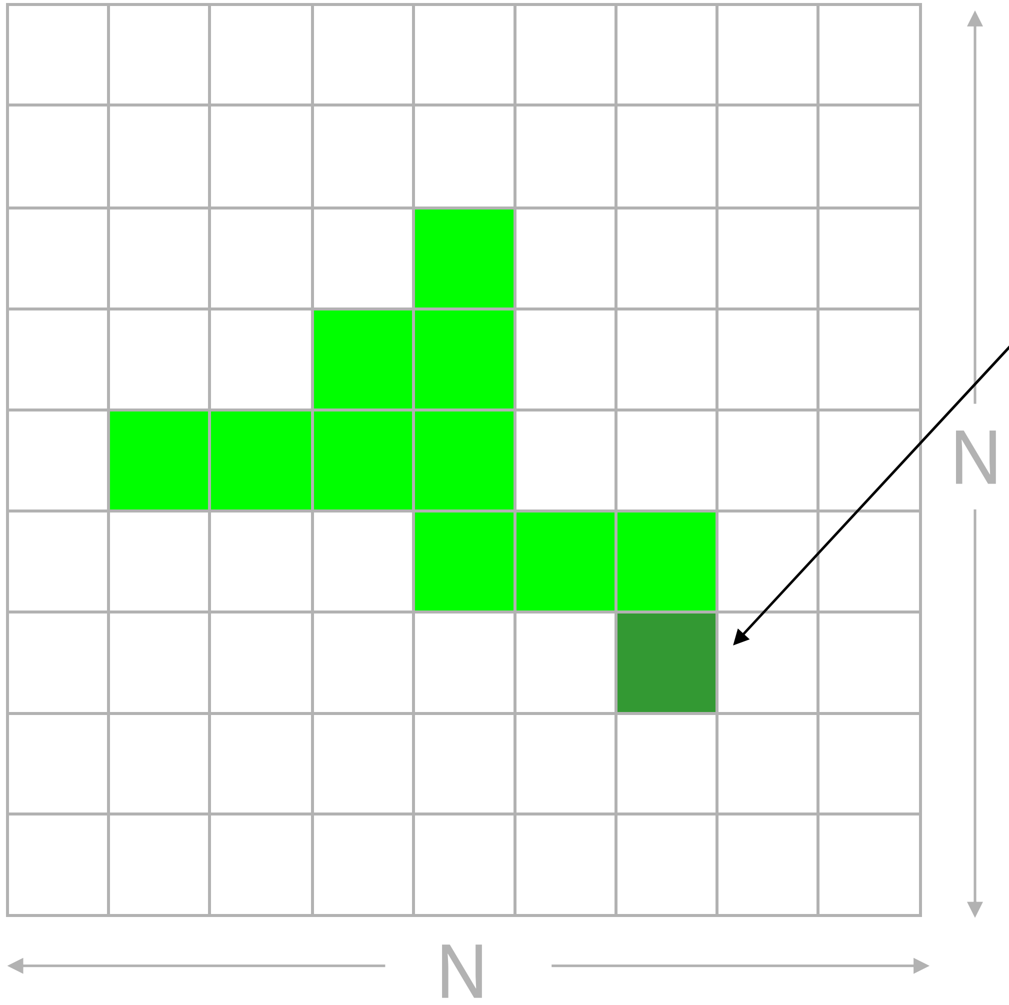
But on average, the filled region is half the size of the grid

$N/2 \cdot N/2 = N^2/4$

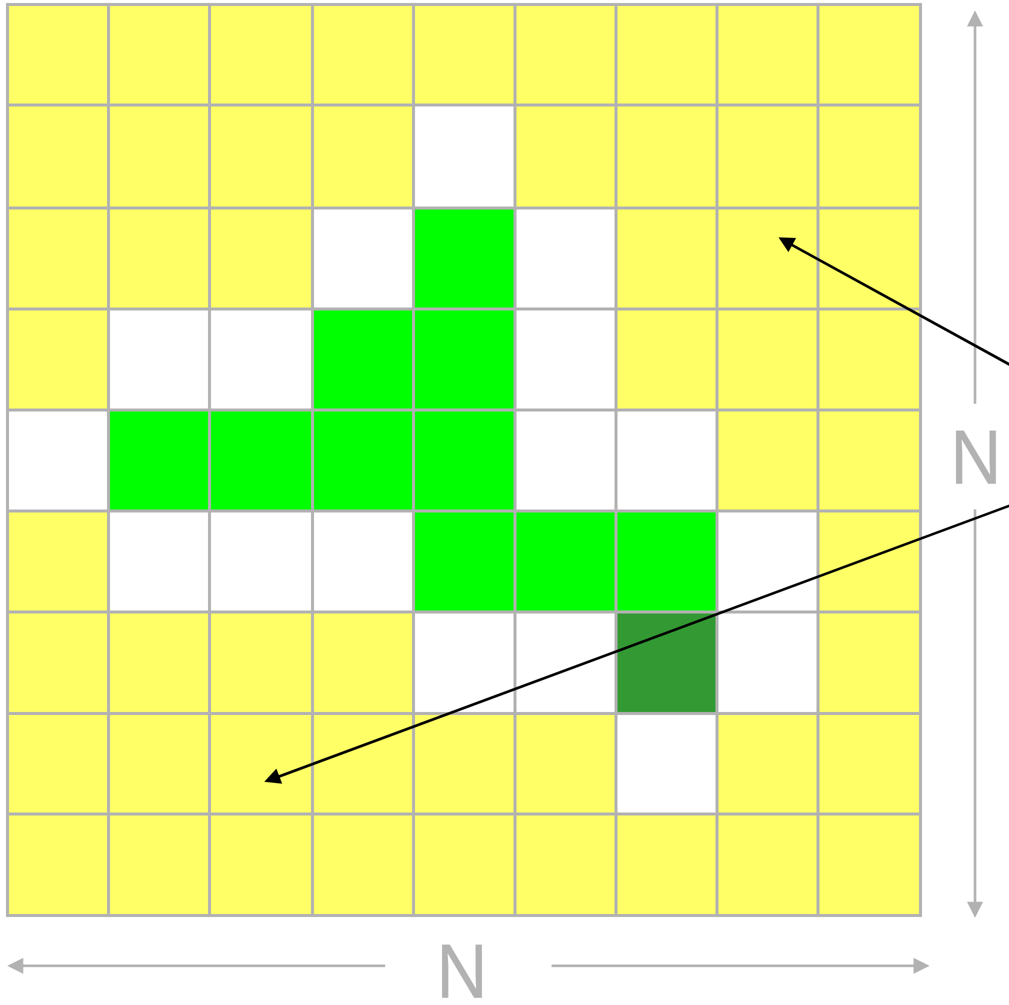
115 days → 29 days

148N×148N grid eats that up

Need to attack the *exponent*

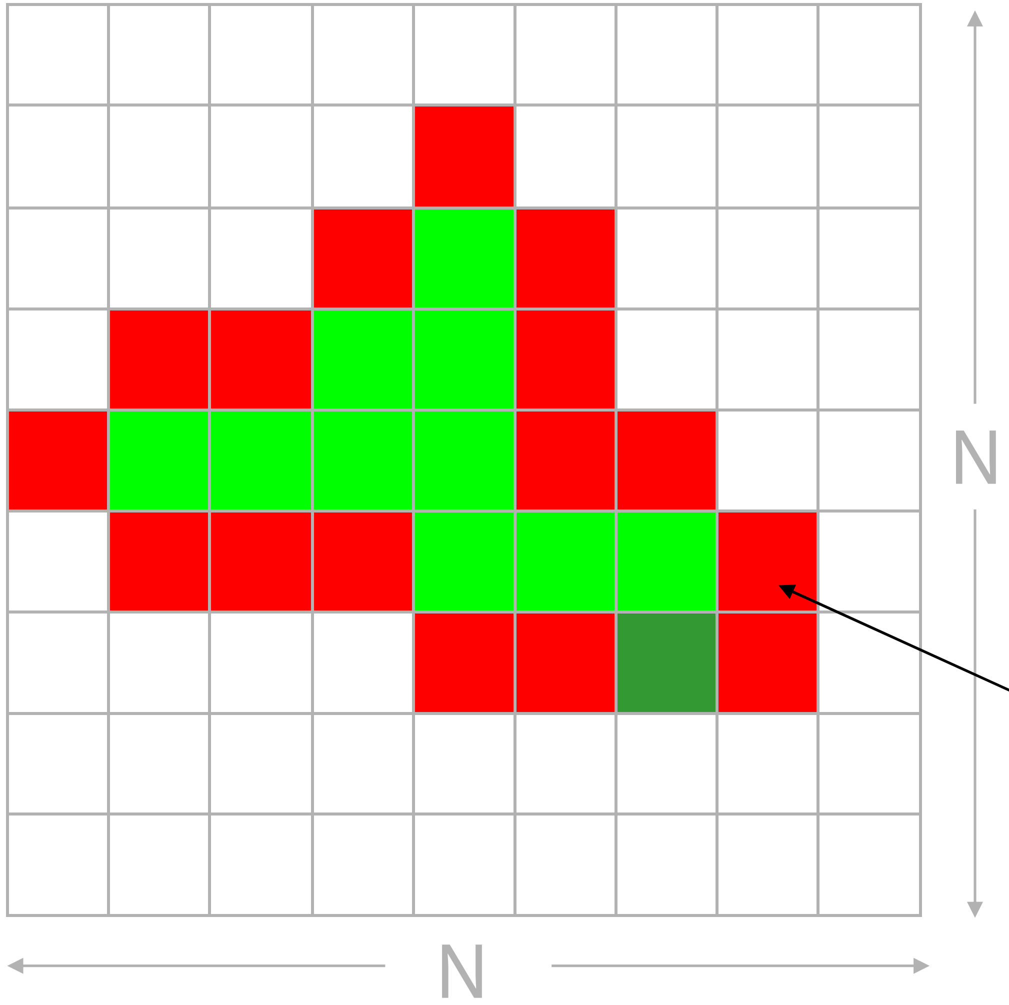


If we just added this cell...



If we just added this cell...

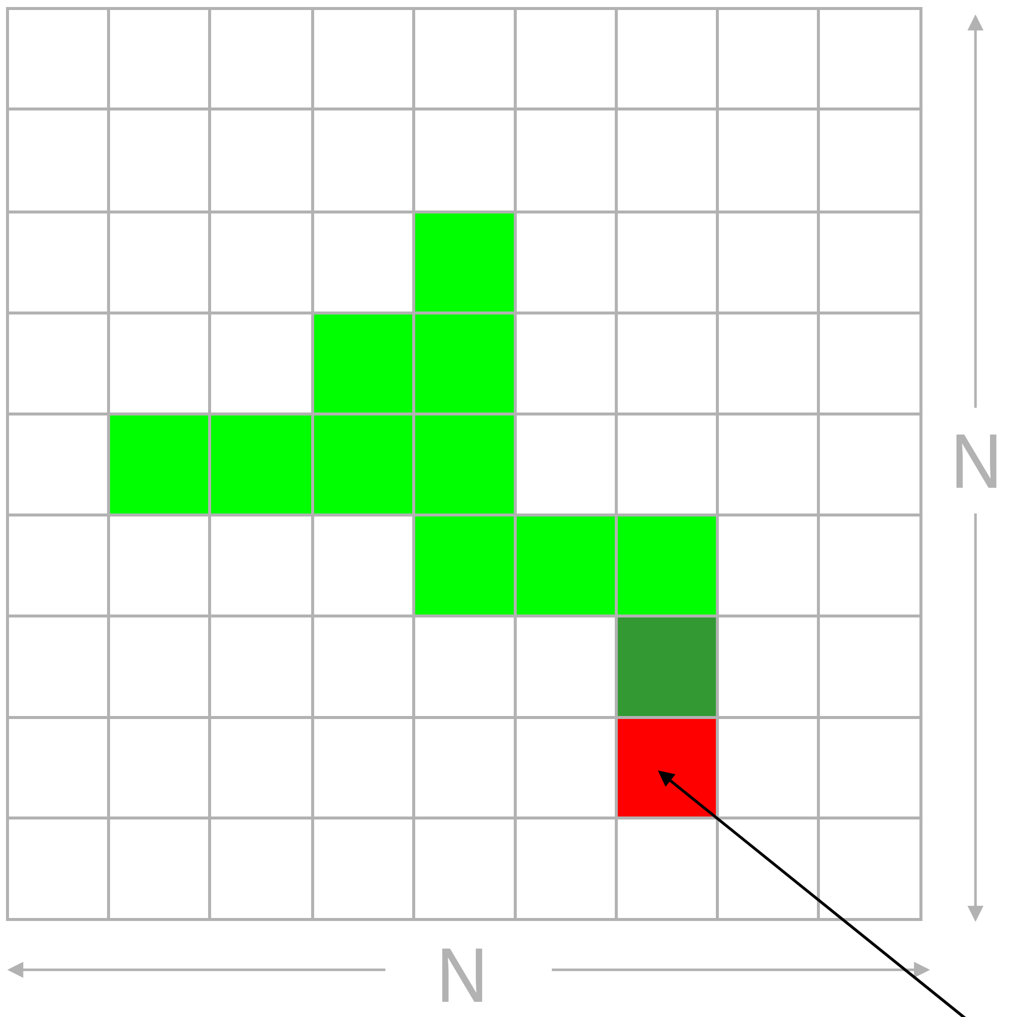
...why check these cells again?



If we just added this cell...

...why check these cells again?

We should already know that these are candidates



If we just added this cell...

...why check these cells again?

We should already know that these are candidates

This is the only new candidate cell

Big idea: trade memory for time

Big idea: trade memory for time

Record redundant information in order to save re-calculation

Big idea: trade memory for time

Record redundant information in order to save re-calculation

In this case, keep a list of cells on the boundary

Big idea: trade memory for time

Record redundant information in order to save re-calculation

In this case, keep a list of cells on the boundary

Each time a cell is filled, check its neighbors

Big idea: trade memory for time

Record redundant information in order to save re-calculation

In this case, keep a list of cells on the boundary

Each time a cell is filled, check its neighbors

If already in the list, do nothing

Big idea: trade memory for time

Record redundant information in order to save re-calculation

In this case, keep a list of cells on the boundary

Each time a cell is filled, check its neighbors

If already in the list, do nothing

Otherwise, add to list

Big idea: trade memory for time

Record redundant information in order to save re-calculation

In this case, keep a list of cells on the boundary

Each time a cell is filled, check its neighbors

If already in the list, do nothing

Otherwise, ~~add~~ to list

Insert into list so that low-valued cells at front

Big idea: trade memory for time

Record redundant information in order to save re-calculation

In this case, keep a list of cells on the boundary

Each time a cell is filled, check its neighbors

If already in the list, do nothing

Otherwise, ~~add~~ to list

Insert into list so that low-valued cells at front

Making it easy to choose next cell to fill

0	1	2	3	4	5	6	7	8	
5	3	7	2	6	1	1	3	4	8
8	5	6	5	7	2	3	6	2	7
2	5	8	7	5	5	6	5	9	6
5	2	6	4	9	3	9	6	5	5
4	6	8	8	5	9	7	3	9	4
7	6	4	5	1	2	6	8	5	3
5	4	2	5	8	5	5	5	8	2
5	7	5	1	5	3	8	5	5	1
4	5	1	9	7	8	6	5	1	0

List of cells on edge is initially empty

```
edge = []
```

	0	1	2	3	4	5	6	7	8	
	5	3	7	2	6	1	1	3	4	8
	8	5	6	5	7	2	3	6	2	7
	2	5	8	7	5	5	6	5	9	6
	5	2	6	4	9	3	9	6	5	5
	4	6	8	8	-1	9	7	3	9	4
	7	6	4	5	1	2	6	8	5	3
	5	4	2	5	8	5	5	5	8	2
	5	7	5	1	5	3	8	5	5	1
	4	5	1	9	7	8	6	5	1	0

List of cells on edge is initially empty

Fill center cell and add its neighbors

```
edge = [(1,4,3), (8,3,4), (9,4,5), (9,5,4)]
```

0	1	2	3	4	5	6	7	8	
5	3	7	2	6	1	1	3	4	8
8	5	6	5	7	2	3	6	2	7
2	5	8	7	5	5	6	5	9	6
5	2	6	4	9	3	9	6	5	5
4	6	8	8	-1	9	7	3	9	4
7	6	4	5	1	2	6	8	5	3
5	4	2	5	8	5	5	5	8	2
5	7	5	1	5	3	8	5	5	1
4	5	1	9	7	8	6	5	1	0

List of cells on edge is initially empty

Fill center cell and add its neighbors

value

```
edge = [(1,4,3), (8,3,4), (9,4,5), (9,5,4)]
```

coordinates

0	1	2	3	4	5	6	7	8	
5	3	7	2	6	1	1	3	4	8
8	5	6	5	7	2	3	6	2	7
2	5	8	7	5	5	6	5	9	6
5	2	6	4	9	3	9	6	5	5
4	6	8	8	-1	9	7	3	9	4
7	6	4	5	-1	2	6	8	5	3
5	4	2	5	8	5	5	5	8	2
5	7	5	1	5	3	8	5	5	1
4	5	1	9	7	8	6	5	1	0

Take first cell from list and fill it

```
edge = [(8,3,4), (9,4,5), (9,5,4)]
```

0	1	2	3	4	5	6	7	8	
5	3	7	2	6	1	1	3	4	8
8	5	6	5	7	2	3	6	2	7
2	5	8	7	5	5	6	5	9	6
5	2	6	4	9	3	9	6	5	5
4	6	8	8	-1	9	7	3	9	4
7	6	4	5	-1	2	6	8	5	3
5	4	2	5	8	5	5	5	8	2
5	7	5	1	5	3	8	5	5	1
4	5	1	9	7	8	6	5	1	0

Take first cell from list and fill it

Add its neighbors to the list

```
edge = [(2,5,3), (5,3,3), (8,4,2), (8,3,4),
        (9,4,5), (9,5,4)]
```

0	1	2	3	4	5	6	7	8	
5	3	7	2	6	1	1	3	4	8
8	5	6	5	7	2	3	6	2	7
2	5	8	7	5	5	6	5	9	6
5	2	6	4	9	3	9	6	5	5
4	6	8	8	-1	9	7	3	9	4
7	6	4	5	-1	-1	6	8	5	3
5	4	2	5	8	5	5	5	8	2
5	7	5	1	5	3	8	5	5	1
4	5	1	9	7	8	6	5	1	0

Take, fill, and add again

```
edge = [(5,3,3), (5,5,2), (6,6,3), (8,4,2), (8,3,4),
          (9,4,5), (9,5,4)]
```

0	1	2	3	4	5	6	7	8	
5	3	7	2	6	1	1	3	4	8
8	5	6	5	7	2	3	6	2	7
2	5	8	7	5	5	6	5	9	6
5	2	6	4	9	3	9	6	5	5
4	6	8	8	-1	9	7	3	9	4
7	6	4	5	-1	-1	6	8	5	3
5	4	2	5	8	5	5	5	8	2
5	7	5	1	5	3	8	5	5	1
4	5	1	9	7	8	6	5	1	0

Take, fill, and add again

Don't re-add cells that are already in the list

```
edge = [(5,3,3), (5,5,2), (6,6,3), (8,4,2), (8,3,4),
        (9,4,5), (9,5,4)]
```

0	1	2	3	4	5	6	7	8	
5	3	7	2	6	1	1	3	4	8
8	5	6	5	7	2	3	6	2	7
2	5	8	7	5	5	6	5	9	6
5	2	6	4	9	3	9	6	5	5
4	6	8	8	-1	9	7	3	9	4
7	6	4	5	-1	-1	6	8	5	3
5	4	2	5	8	5	5	5	8	2
5	7	5	1	5	3	8	5	5	1
4	5	1	9	7	8	6	5	1	0

In case of ties, find all cells at the front of the list with the lowest value...

```
edge = [(5,3,3), (5,5,2), (6,6,3), (8,4,2), (8,3,4),
        (9,4,5), (9,5,4)]
```

0	1	2	3	4	5	6	7	8	
5	3	7	2	6	1	1	3	4	8
8	5	6	5	7	2	3	6	2	7
2	5	8	7	5	5	6	5	9	6
5	2	6	4	9	3	9	6	5	5
4	6	8	8	-1	9	7	3	9	4
7	6	4	5	-1	-1	6	8	5	3
5	4	2	5	8	-1	5	5	8	2
5	7	5	1	5	3	8	5	5	1
4	5	1	9	7	8	6	5	1	0

In case of ties, find all cells at the front of the list with the lowest value...

...and select and fill one of them

```
edge = [(3,5,1), (5,3,3), (5,5,2), (6,6,3), (8,4,2),
        (8,3,4), (9,4,5), (9,5,4)]
```

How quickly can we insert into a sorted list?

How quickly can we insert into a sorted list?

```
def insert(values, new_val):
  '''Insert (v, x, y) tuple into list in right place.'''
  i = 0
  while i < len(values):
    if values[i][0] >= new_val[0]:
      break
  values.insert(i, new_val)
```

How quickly can we insert into a sorted list?

```
def insert(values, new_val):
  '''Insert (v, x, y) tuple into list in right place.'''
  i = 0
  while i < len(values):
    if values[i][0] >= new_val[0]:
      break
  values.insert(i, new_val)
```

Works even if list is empty…

How quickly can we insert into a sorted list?

```
def insert(values, new_val):
  '''Insert (v, x, y) tuple into list in right place.'''
  i = 0
  while i < len(values):
    if values[i][0] >= new_val[0]:
      break
  values.insert(i, new_val)
```

Works even if list is empty...

...or new value greater than all values in list

If list has K values…

If list has K values...

...insertion takes on average K/2 steps

If list has K values...

...insertion takes on average K/2 steps

Our fractals fill about $N^{1.5}$ cells...

If list has K values...

...insertion takes on average K/2 steps

Our fractals fill about $N^{1.5}$ cells...

...and there are as many cells on the boundary as there are in the fractal...

If list has K values...

...insertion takes on average K/2 steps

Our fractals fill about $N^{1.5}$ cells...

...and there are as many cells on the boundary

as there are in the fractal...

...so this takes $N^{1.5} \cdot N^{1.5} = N^3$ steps

If list has K values...

...insertion takes on average K/2 steps

Our fractals fill about $N^{1.5}$ cells...

...and there are as many cells on the boundary as there are in the fractal...

...so this takes $N^{1.5} \cdot N^{1.5} = N^3$ steps

Not much of an improvement on $N^{3.5}$

If list has K values...

...insertion takes on average K/2 steps

Our fractals fill about $N^{1.5}$ cells...

...and there are as many cells on the boundary

as there are in the fractal...

...so this takes $N^{1.5} \cdot N^{1.5} = N^3$ steps

Not much of an improvement on $N^{3.5}$

But there's a much faster way to insert

To look up a name in the phone book...

To look up a name in the phone book...

...open it in the middle

To look up a name in the phone book...

...open it in the middle

If the name there comes after the one you want,

go to the middle of the first half

To look up a name in the phone book...

...open it in the middle

If the name there comes after the one you want,

go to the middle of the first half

If it comes after the one you want, go to the middle of the second half

To look up a name in the phone book...

...open it in the middle

If the name there comes after the one you want, go to the middle of the first half

If it comes after the one you want, go to the middle of the second half

Then repeat this procedure in that half

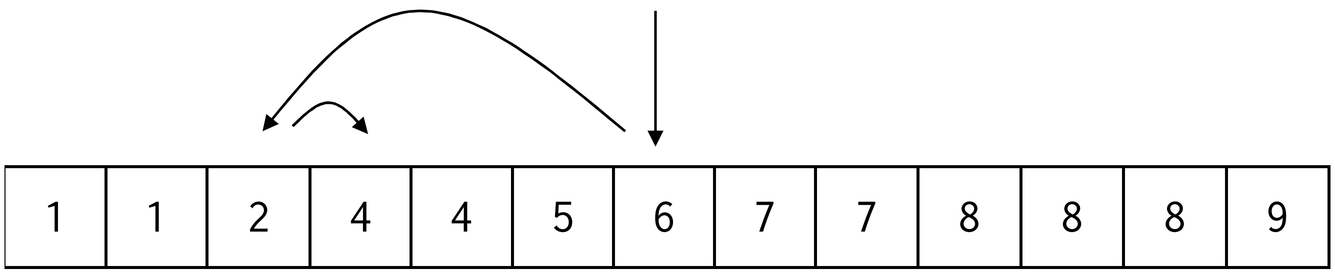
To look up a name in the phone book...

...open it in the middle

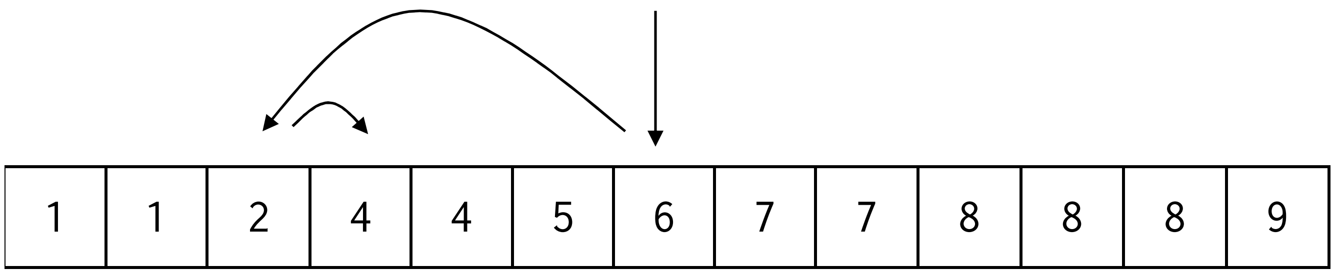
If the name there comes after the one you want, go to the middle of the first half

If it comes after the one you want, go to the middle of the second half

Then repeat this procedure in that half

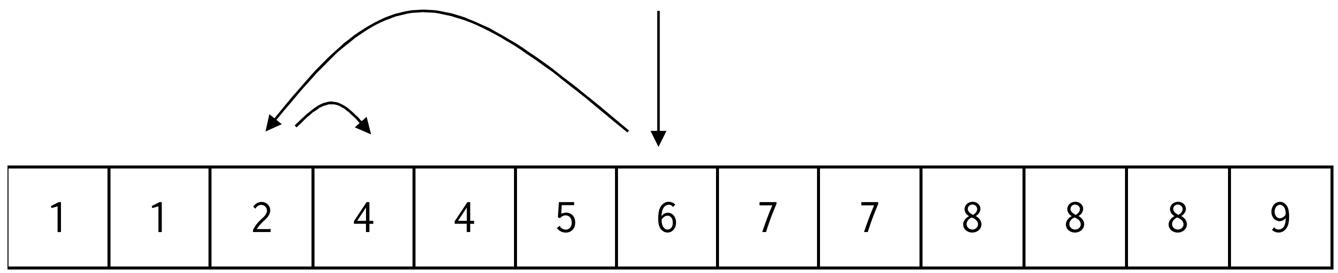


How fast is this?



How fast is this?

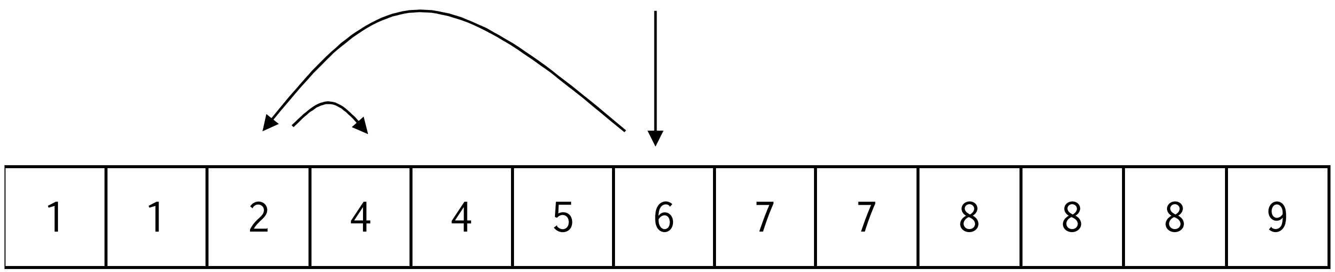
One probe can find one value



How fast is this?

One probe can find one value

Two probes can find one value among two (2^1)

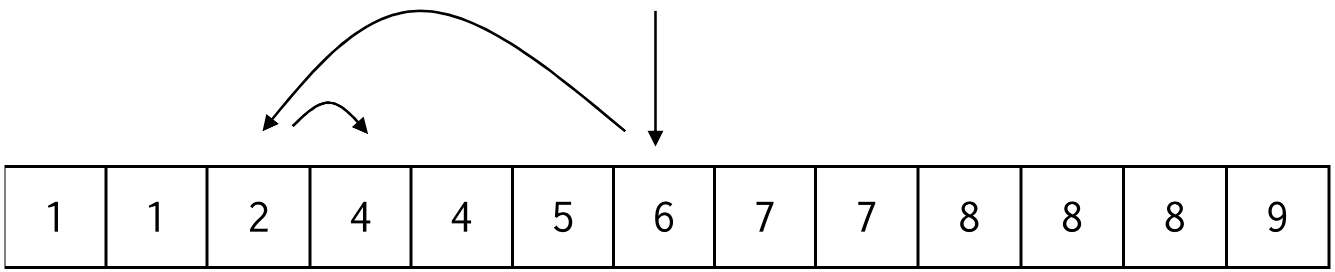


How fast is this?

One probe can find one value

Two probes can find one value among two (2^1)

Three probes can find one value among four (2^2)



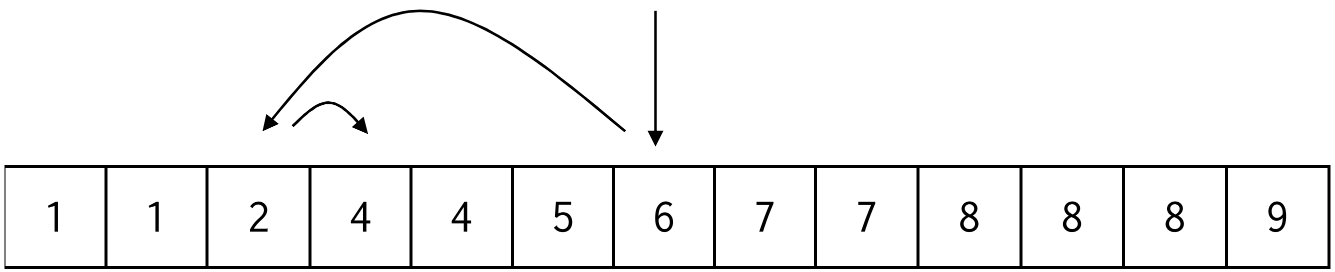
How fast is this?

One probe can find one value

Two probes can find one value among two (2^1)

Three probes can find one value among four (2^2)

Four probes: one among eight (2^3)



How fast is this?

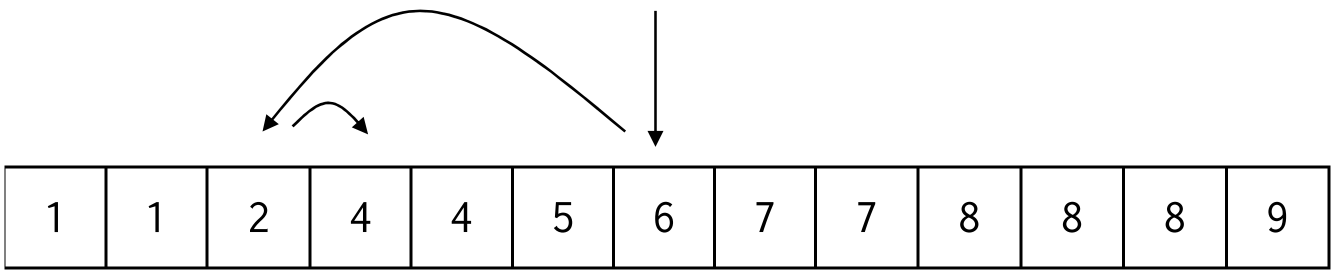
One probe can find one value

Two probes can find one value among two (2^1)

Three probes can find one value among four (2^2)

Four probes: one among eight (2^3)

K probes: one among 2^K



How fast is this?

One probe can find one value

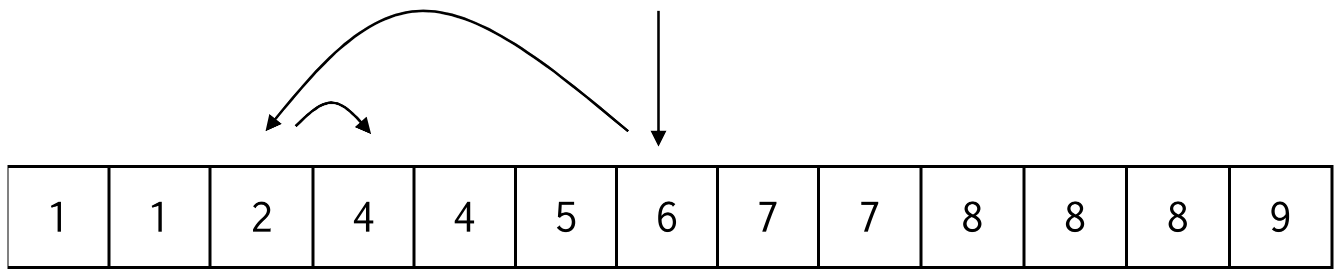
Two probes can find one value among two (2^1)

Three probes can find one value among four (2^2)

Four probes: one among eight (2^3)

K probes: one among 2^K

$\log_2(N)$ probes: one among N values



0	1	2	3	4	5	6	7	8	
5	3	7	2	6	1	1	3	4	8
8	5	6	5	7	2	3	6	2	7
2	5	8	7	5	5	6	5	9	6
5	2	6	4	9	3	9	6	5	5
4	6	8	8	-1	9	7	3	9	4
7	6	4	5	-1	-1	6	8	5	3
5	4	2	5	8	-1	5	5	8	2
5	7	5	1	5	3	8	5	5	1
4	5	1	9	7	8	6	5	1	0

Running time is

$N^{1.5} \cdot \log_2(N^{1.5})$

1	1	2	4	4	5	6	7	7	8	8	8	9

0	1	2	3	4	5	6	7	8	
5	3	7	2	6	1	1	3	4	8
8	5	6	5	7	2	3	6	2	7
2	5	8	7	5	5	6	5	9	6
5	2	6	4	9	3	9	6	5	5
4	6	8	8	-1	9	7	3	9	4
7	6	4	5	-1	-1	6	8	5	3
5	4	2	5	8	-1	5	5	8	2
5	7	5	1	5	3	8	5	5	1
4	5	1	9	7	8	6	5	1	0

Running time is

$N^{1.5} \cdot \log_2(N^{1.5})$

Or $N^{1.5}\log_2(N)$ if we get rid of constants

1	1	2	4	4	5	6	7	7	8	8	8	9

That changes things quite a bit

Grid Size	Old Time	Which Was...	New Time	Which Is...
N	T	1 sec		
2N	11.3 T	11 sec	2.8 T	3 sec
3N	46.7 T	47 sec	8.2 T	8 sec
4N	128 T	2 minutes	16 T	16 sec
10N	3162 T	52 minutes	105 T	2 min
50N	883883 T	1 day	1995 T	33 min
100N	10^7 T	115 days	6644 T	2 hours

That changes things quite a bit

Grid Size	Old Time	Which Was...	New Time	Which Is...
N	T	1 sec		
2N	11.3 T	11 sec	2.8 T	3 sec
3N	46.7 T	47 sec	8.2 T	8 sec
4N	128 T	2 minutes	16 T	16 sec
10N	3162 T	52 minutes	105 T	2 min
50N	883883 T	1 day	1995 T	33 min
100N	10^7 T	**115 days**	6644 T	**2 hours**

That changes things quite a bit

And the gain gets bigger as N increases

Grid Size	Old Time	Which Was...	New Time	Which Is...
N	T	1 sec		
2N	11.3 T	11 sec	2.8 T	3 sec
3N	46.7 T	47 sec	8.2 T	8 sec
4N	128 T	2 minutes	16 T	16 sec
10N	3162 T	52 minutes	105 T	2 min
50N	883883 T	1 day	1995 T	33 min
100N	10^7 T	**115 days**	6644 T	**2 hours**

"Divide and conquer" technique used to insert values into list is called *binary search*

"Divide and conquer" technique used to insert values into list is called *binary search*

Only works if list values are sorted

"Divide and conquer" technique used to insert values into list is called *binary search*

Only works if list values are sorted

But it keeps the list values sorted

"Divide and conquer" technique used to insert values into list is called *binary search*

Only works if list values are sorted

But it keeps the list values sorted

Python implementation is in `bisect` library

We can do even better

We can do even better

5	3	7	2	6	1	1	3	4
8	5	6	5	7	2	3	6	2
2	5	8	7	5	5	6	5	9
5	2	6	4	9	3	9	6	5
4	6	8	8	5	9	7	3	9
7	6	4	5	1	2	6	8	5
5	4	2	5	8	5	5	5	8
5	7	5	1	5	3	8	5	5
4	5	1	9	7	8	6	5	1

Generating the grid

takes N^2 steps

We can do even better

5	3	7	2	6	1	1	3	4
8	5	6	5	7	2	3	6	2
2	5	8	7	5	5	6	5	9
5	2	6	4	9	3	9	6	5
4	6	8	8	−1	9	7	3	9
7	6	4	5	−1	−1	6	8	5
5	4	2	5	8	−1	−1	5	8
5	7	−1	−1	−1	−1	8	5	5
4	5	−1	9	7	8	6	5	1

Generating the grid takes N^2 steps

If we fill these cells...

We can do even better

5	3	7	2	6	1	1	3	4
8	5	6	5	7	2	3	6	2
2	5	8	7	5	5	6	5	9
5	2	6	4	9	3	9	6	5
4	6	8	8	−1	9	7	3	9
7	6	4	5	−1	−1	6	8	5
5	4	2	5	8	−1	−1	5	8
5	7	−1	−1	−1	−1	8	5	5
4	5	−1	9	7	8	6	5	1

Generating the grid

takes N^2 steps

If we fill these

cells...

...we only ever look

at these cells...

We can do even better

				9				
			8	−1	9			
			5	−1	−1	6		
		2	5	8	−1	−1	5	
	7	−1	−1	−1	−1	8		
	5	−1	9	7	8	6		

Generating the grid takes N^2 steps

If we fill these cells...

...we only ever look at these cells...

...so why bother generating values for these ones?

Store grid as a dictionary

Store grid as a dictionary

Keys are (x,y) coordinates of cells

Store grid as a dictionary

Keys are (x,y) coordinates of cells

Values are current cell values

Store grid as a dictionary

Keys are (x,y) coordinates of cells

Values are current cell values

Instead of `grid[x][y]`, use `get_value(grid, x, y, Z)`

Store grid as a dictionary

Keys are (x,y) coordinates of cells

Values are current cell values

Instead of `grid[x][y]`, use `get_value(grid, x, y, Z)`

```
def get_value(grid, x, y, Z):
  '''Get value of grid cell, creating if necessary.'''
  if (x, y) not in grid:
    grid[(x, y)] = random.randint(1, Z)
  return grid[(x, y)]
```

Store grid as a dictionary

Keys are (x,y) coordinates of cells

Values are current cell values

Instead of `grid[x][y]`, use `get_value(grid, x, y, Z)`

```
def get_value(grid, x, y, Z):
  '''Get value of grid cell, creating if necessary.'''
  if (x, y) not in grid:
    grid[(x, y)] = random.randint(1, Z)
  return grid[(x, y)]
```

And of course use `set_value(grid, x, y, V)` as well

Another common optimization

Another common optimization

Lazy evaluation

Another common optimization

Lazy evaluation

Don't compute values until they're actually needed

Another common optimization

Lazy evaluation

Don't compute values until they're actually needed

Again, makes program more complicated...

Another common optimization

Lazy evaluation

Don't compute values until they're actually needed

Again, makes program more complicated...

...but also faster

Another common optimization

Lazy evaluation

Don't compute values until they're actually needed

Again, makes program more complicated...

...but also faster

Trading human time for machine performance

How much work does this save us?

How much work does this save us?

Old cost of creating grid: N^2

How much work does this save us?

Old cost of creating grid: N^2

New cost: roughly $N^{1.5}$

How much work does this save us?

Old cost of creating grid: N^2

New cost: roughly $N^{1.5}$

Difference is *not* $N^{0.5}$

How much work does this save us?

Old cost of creating grid: N^2

New cost: roughly $N^{1.5}$

Difference is *not* $N^{0.5}$

As N gets large, $N^2 - N^{1.5} \approx N^2$

How much work does this save us?

Old cost of creating grid: N^2

New cost: roughly $N^{1.5}$

Difference is *not* $N^{0.5}$

As N gets large, $N^2 - N^{1.5} \approx N^2$

I.e., without this change, total runtime would be

$N^2 + N^{1.5}\log_2(N) \approx N^2$

Moral of the story:

Moral of the story:

Biggest performance gains always come from changing algorithms and data structures

The story's other moral:

The story's other moral:

Write and test a simple version first,

then improve it piece by piece

(re-using the tests to check your work)

created by

Greg Wilson

June 2010