CSCI 132:
Basic Data Structures and Algorithms

Sorting (Part 4)

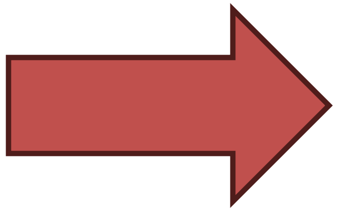
Reese Pearsall + Iliana Castillon

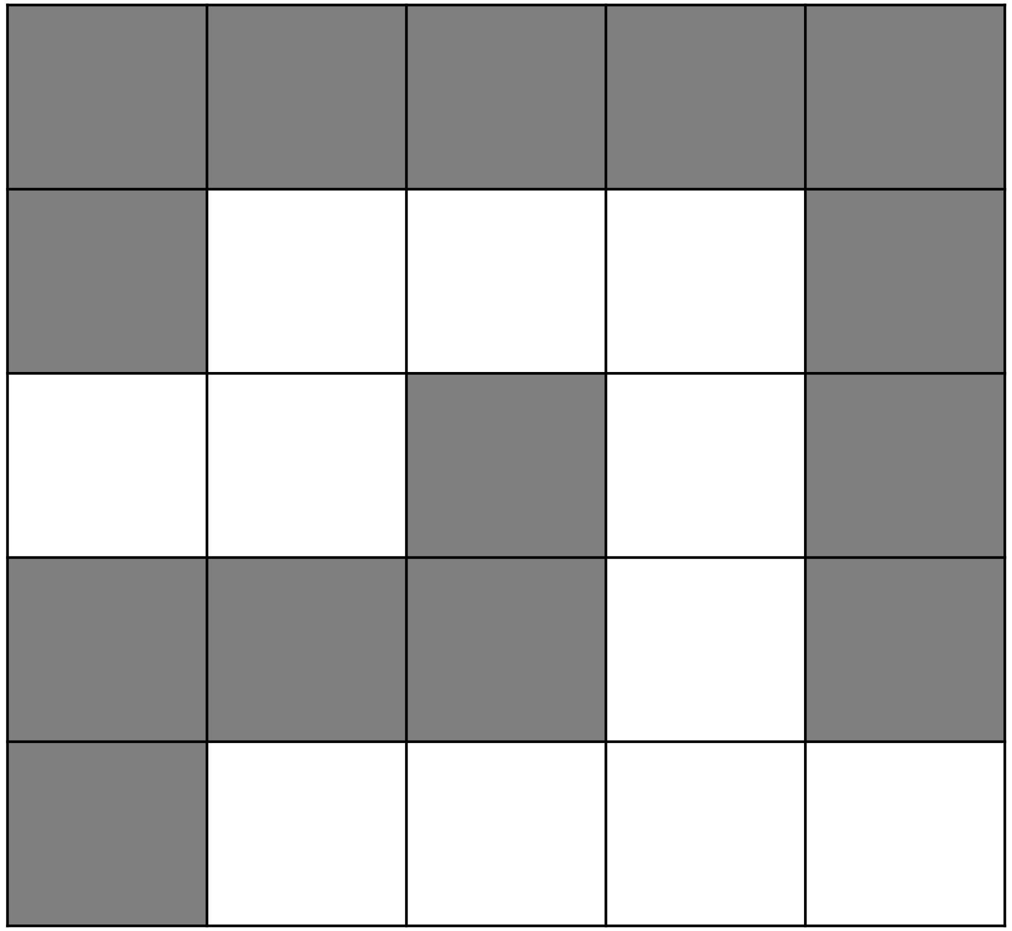
Fall 2024

Announcements

- Friday will be a workday (no lecture)
- Lab 12 due tomorrow @ 11:59PM
- Program 5 posted, Sunday due 12/8
- Rubber duck extra credit will be posted soon

```
char[][] maze

[ [ #, #, #, # ,#],
  [ #, . , . , . , #],
  [ . , . ,# , . , #],
  [ #, #, #, . , #],
  [ #, . , . , . , . ],
 ]
```

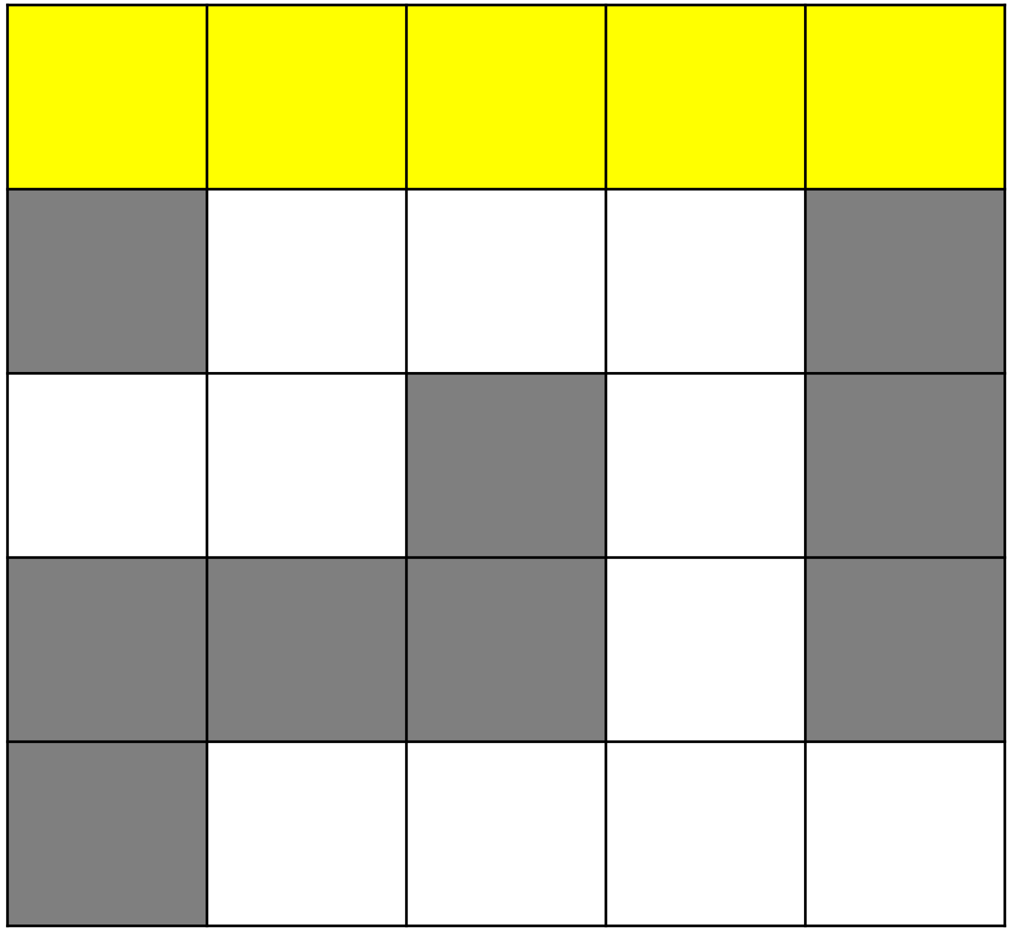




```
char[][] maze
```

```
[ [ #, #, #, # ,#],
  [ #, . , . , . , #],
  [ . , . ,# , . , #],
  [ #, #, #, . , #],
  [ #, . , . , . , . ],
 ]
```

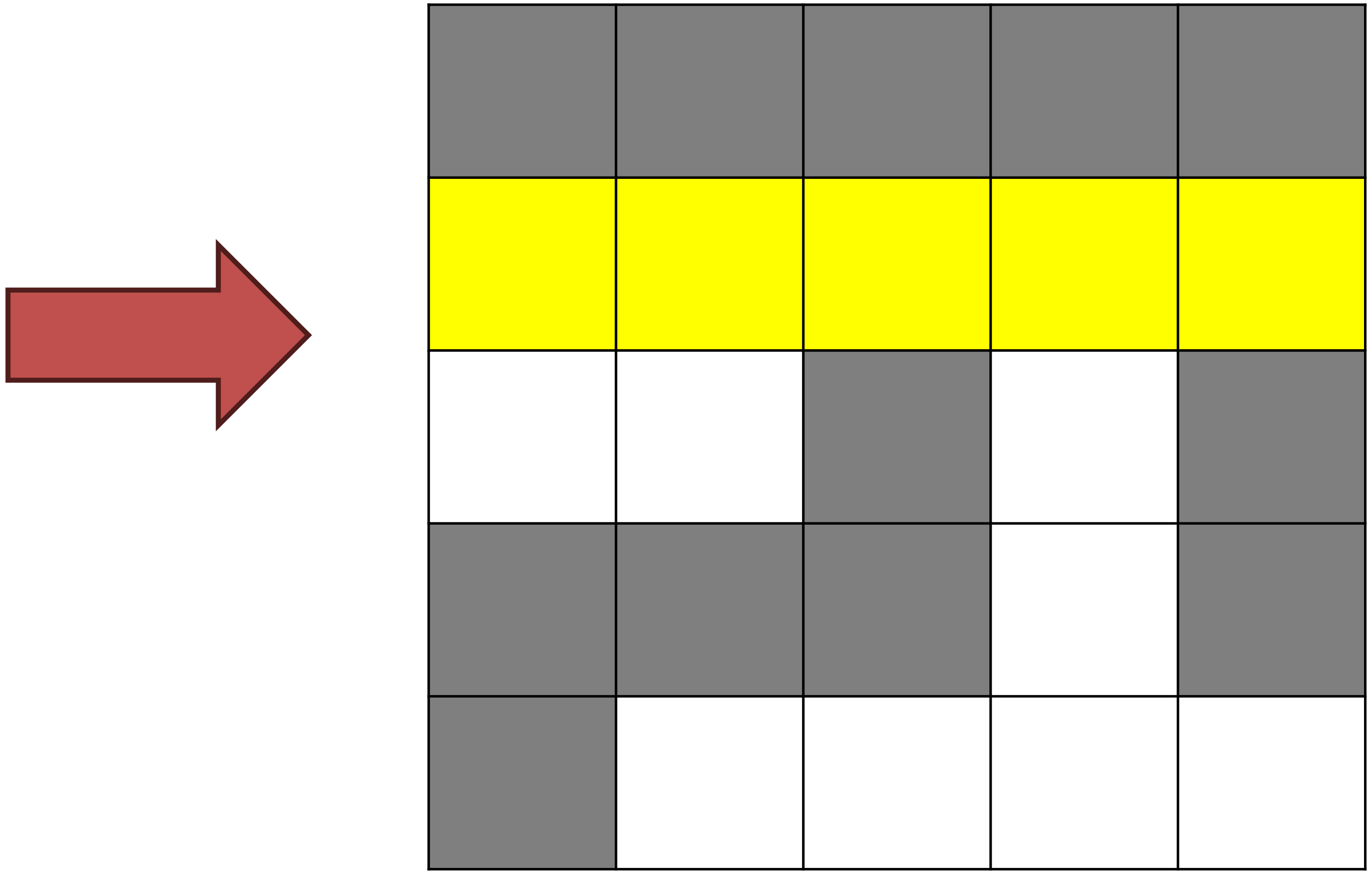
```
maze[0]
```



```
char[][] maze
```

```
[ [ #, #, #, # ,#],
  [ #, . , . , . , #],
  [ . , . ,# , . , #],
  [ #, #, #, . , #],
  [ #, . , . , . , . ],
]
```

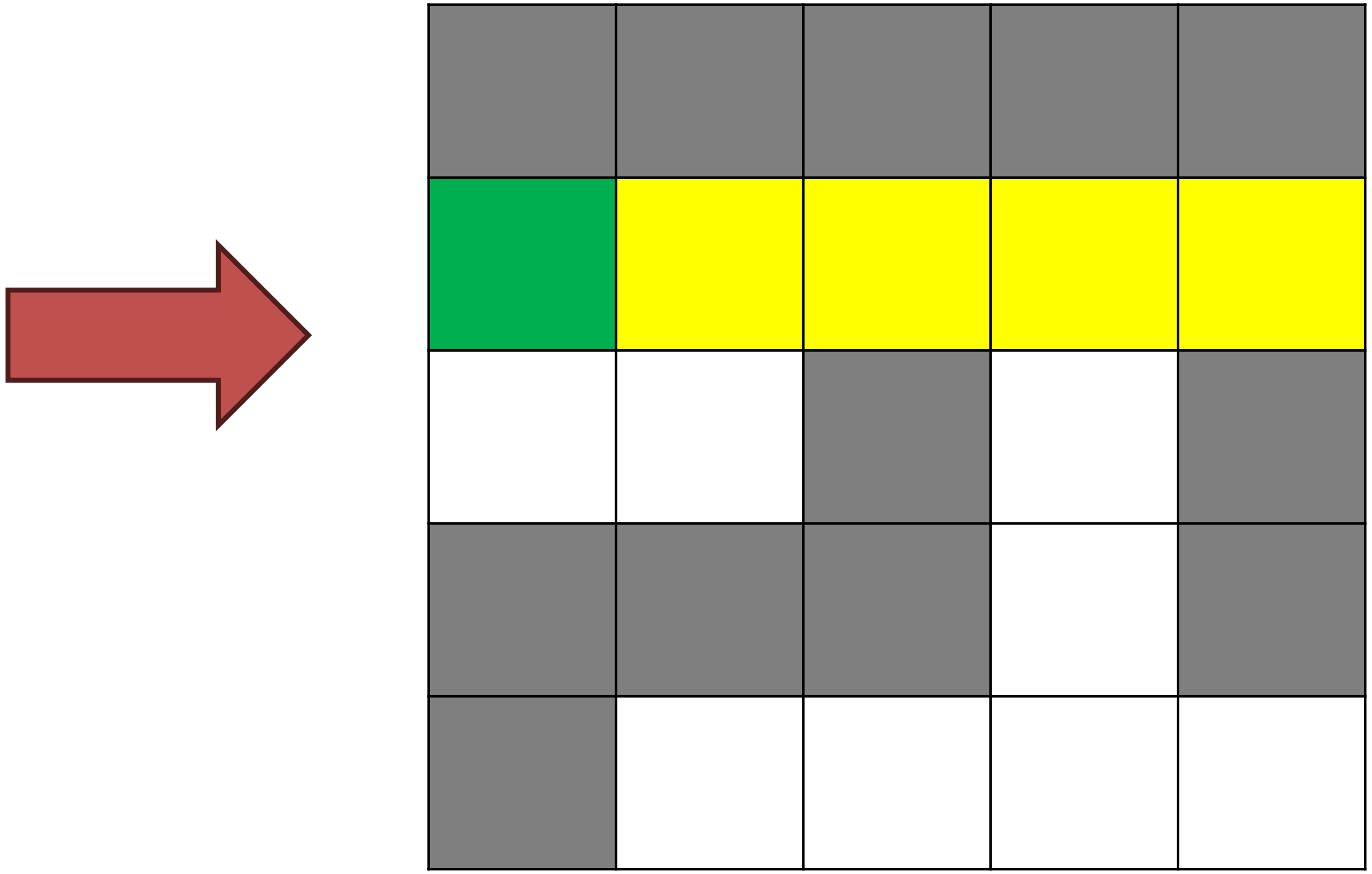
```
maze[1]
```



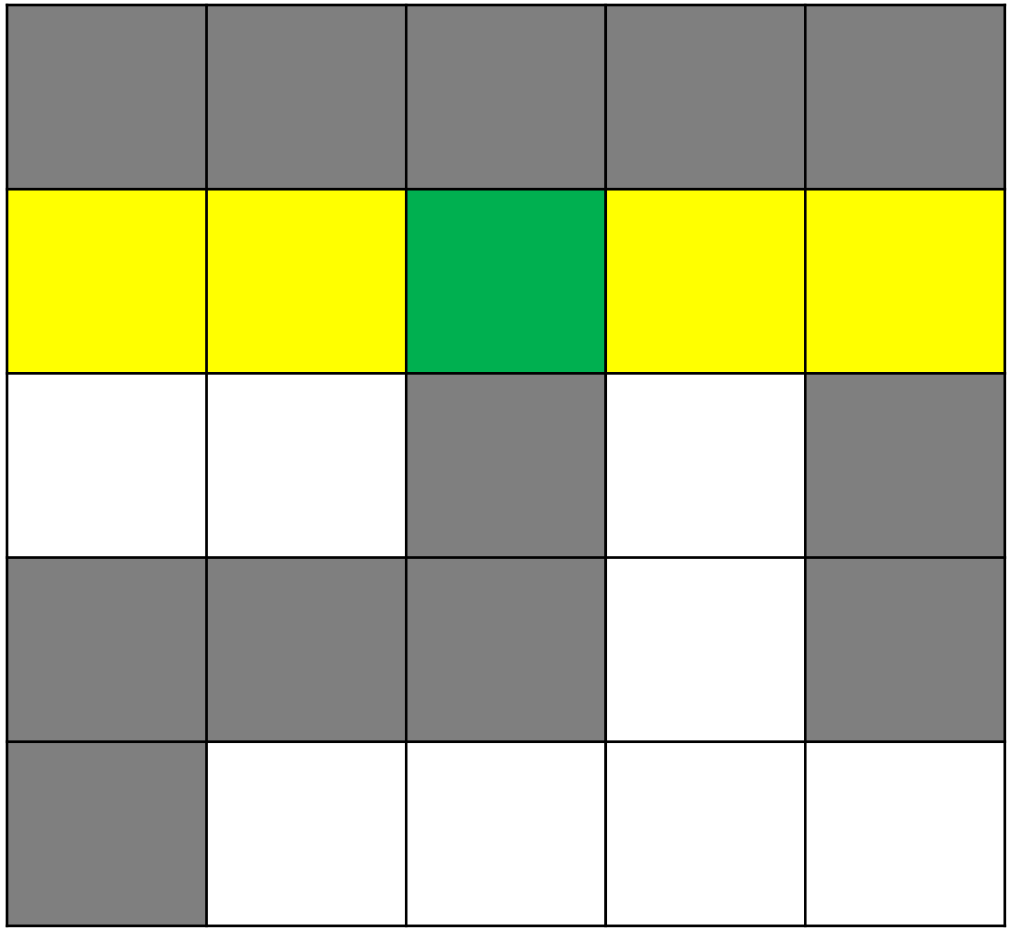
```
char[][] maze
```

```
[ [ #, #, #, # ,#],
  [ #, . , . , . , #],
  [ . , . ,# , . , #],
  [ #, #, #, . , #],
  [ #, . , . , . , . ],
 ]
```

```
maze[1][0]
```



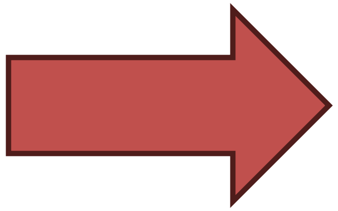
```
char[][] maze

[ [ #, #, #, # ,#],
  [ #, . , . , . , #],
  [ . , . ,# , . , #],
  [ #, #, #, . , #],
  [ #, . , . , . , . ],
 ]

maze[1][2]
```

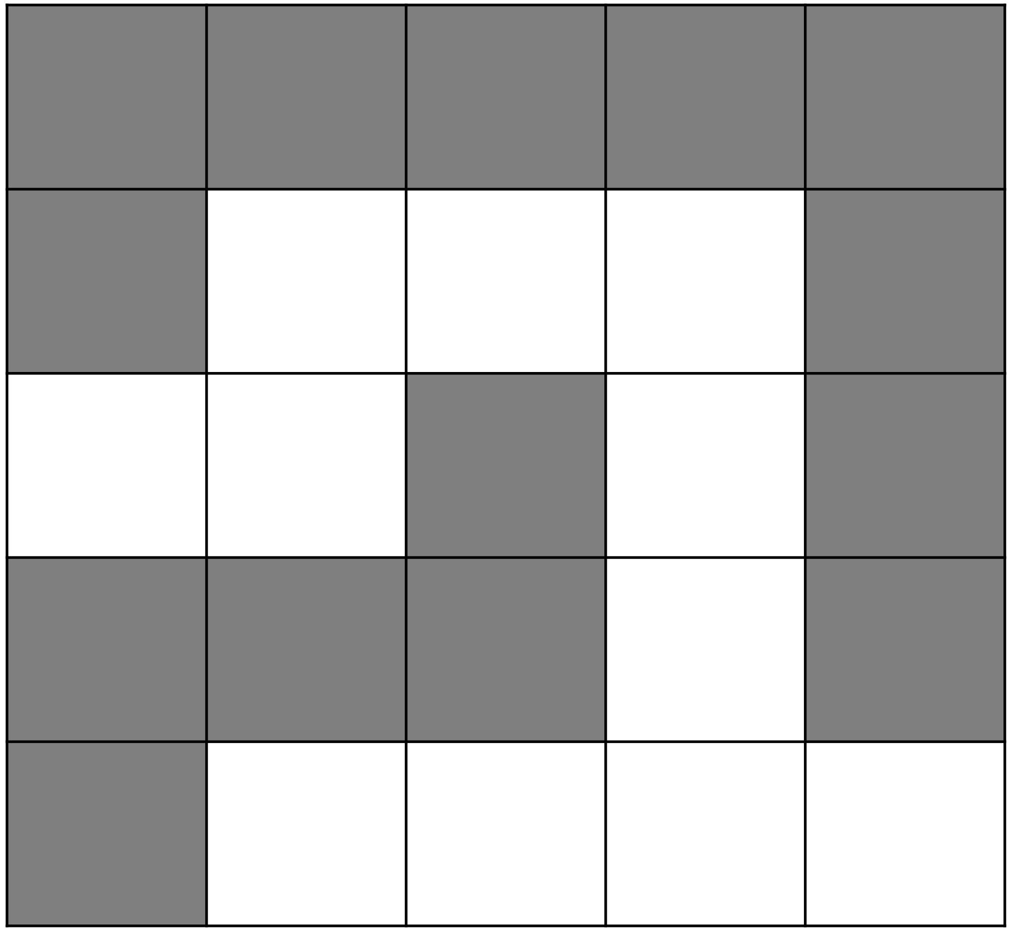


`char[][] maze`

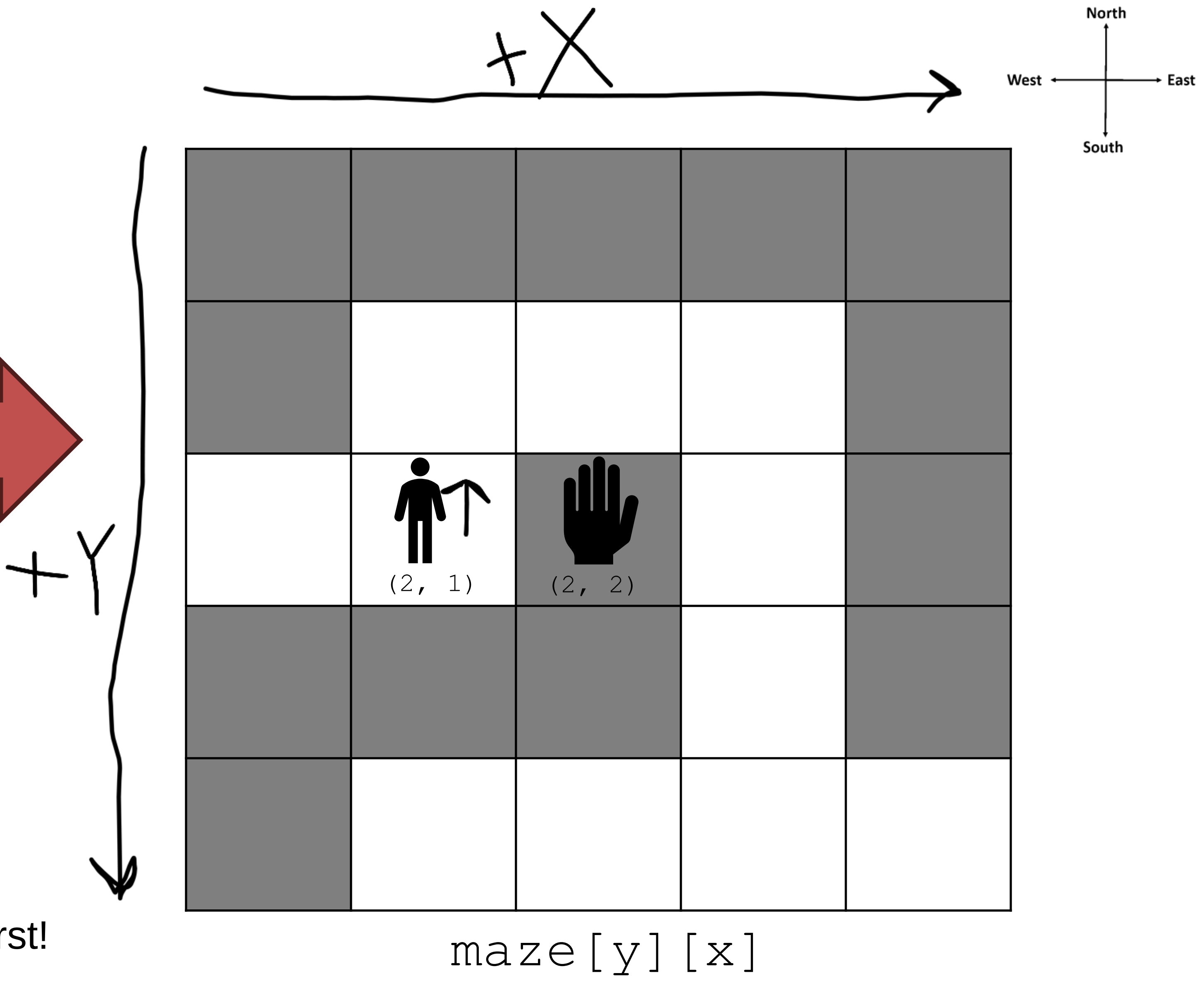
```
[ [ #, #, #, # ,#],
  [ #, . , . , . , #],
  [ . , . ,# , . , #],
  [ #, #, #, . , #],
  [ #, . , . , . , . ],
 ]
```

`maze[y][x]`





```
char[][] maze

[ [ #, #, #, # ,#],
  [ #, . , . , . , #],
  [ . , . ,# , . , #],
  [ #, #, #, . , #],
  [ #, . , . , . , . ],
 ]
```



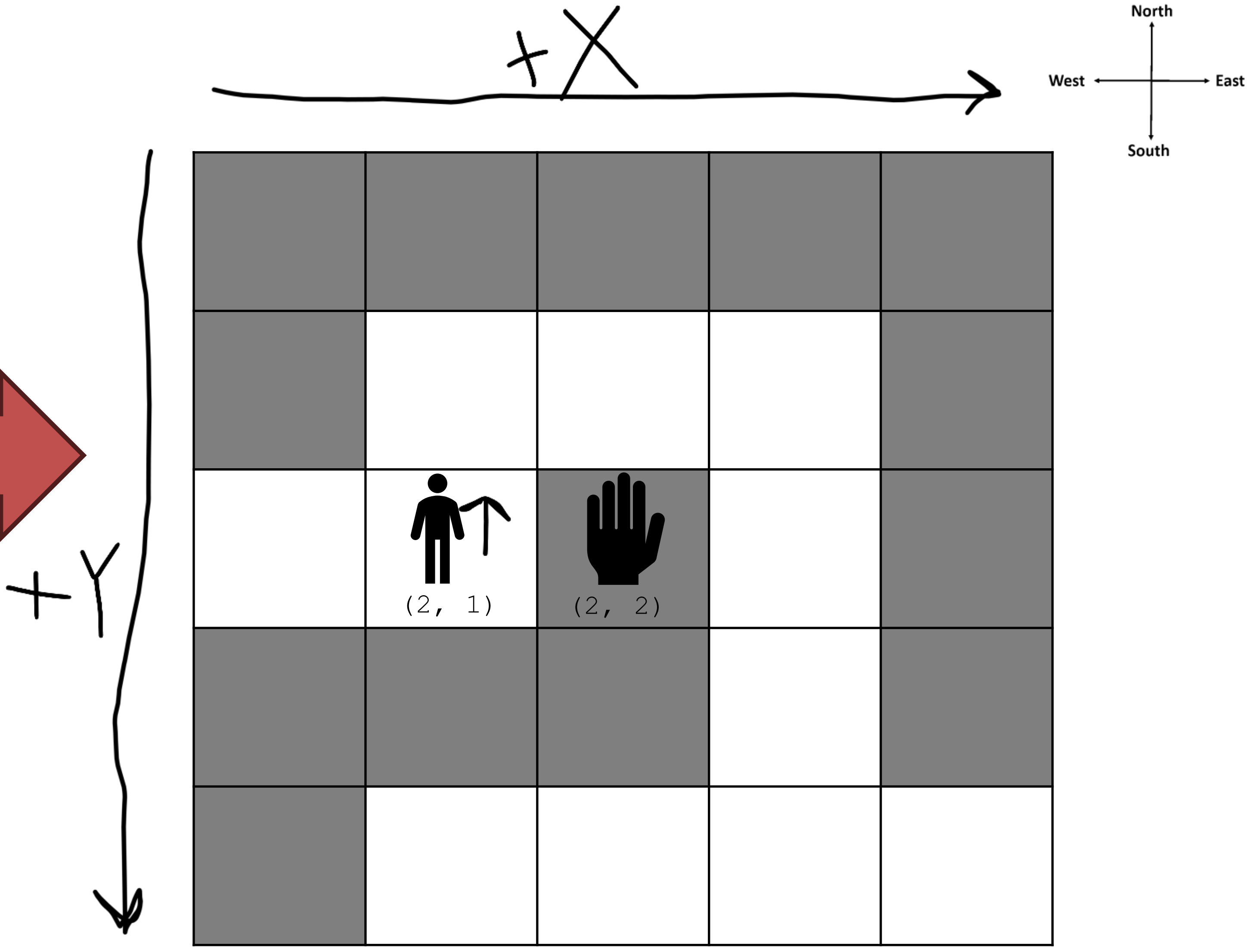
Goal: Move forward one spot

We need to know which direction we are facing first!

How do we know direction we are facing?

char[][] maze

```
[ [ #, #, #, # ,#],
  [ #, . , . , . , #],
  [ . , . ,# , . , #],
  [ #, #, #, . , #],
  [ #, . , . , . , . ],
 ]
```



maze[y][x]

Goal: Move forward one spot

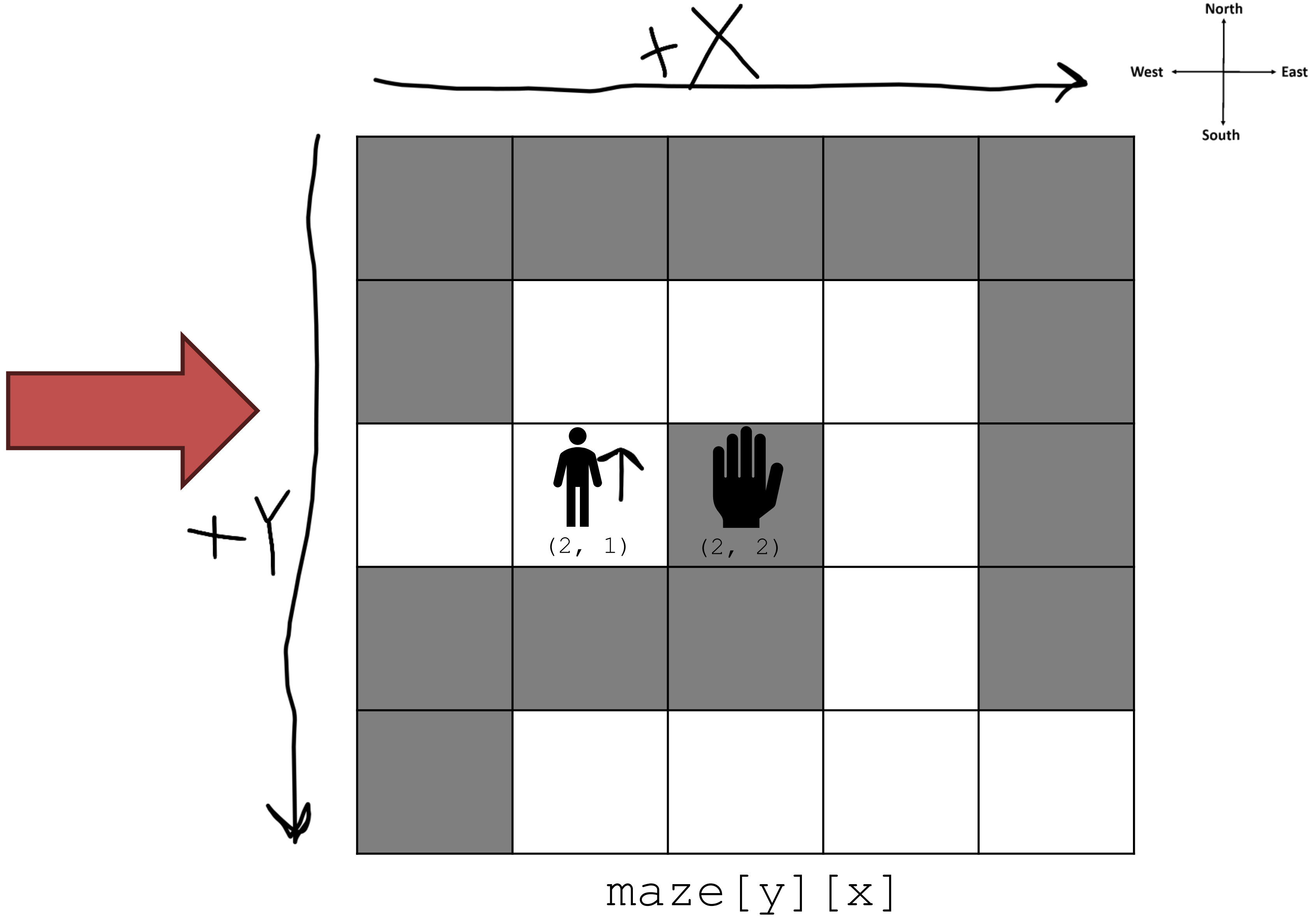
We need to know which direction we are facing first!

Our character Y value and our hand's Y value is the same,
And our character's X value is *less than* our hands' X value

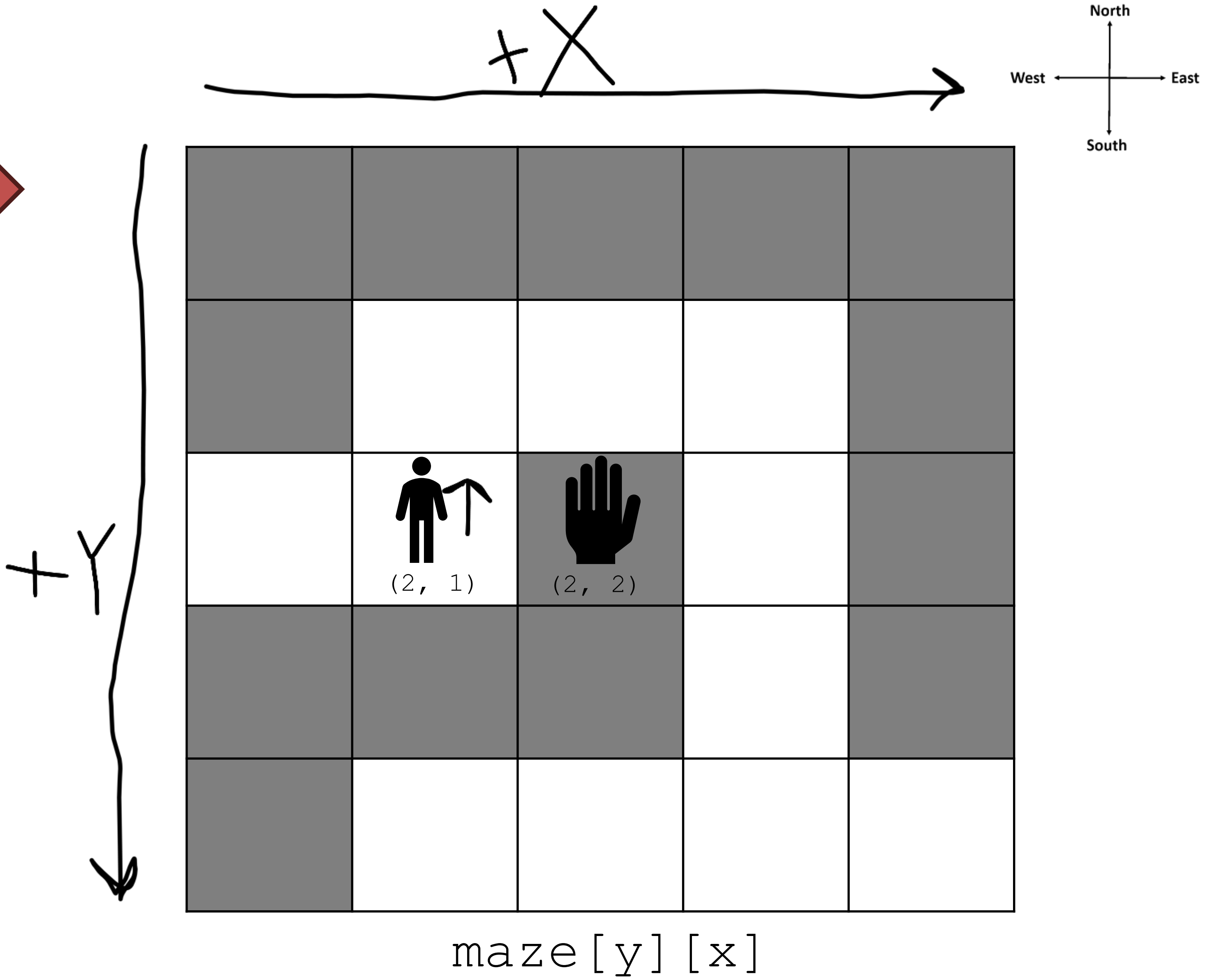
char[][] maze

```
[ [ #, #, #, # ,#],
  [ #, . , . , . , #],
  [ . , . ,# , . , #],
  [ #, #, #, . , #],
  [ #, . , . , . , . ],
 ]
```

```
if(y == hand_y && hand_x > x)
        direction = "North";
}
```



```
char[][] maze

[[#, #, #, # ,#],
 [#, . , . , . , #],
 [. , . ,# , . , #],
 [#, #, #, . , #],
 [#, . , . , . , .],
]
```

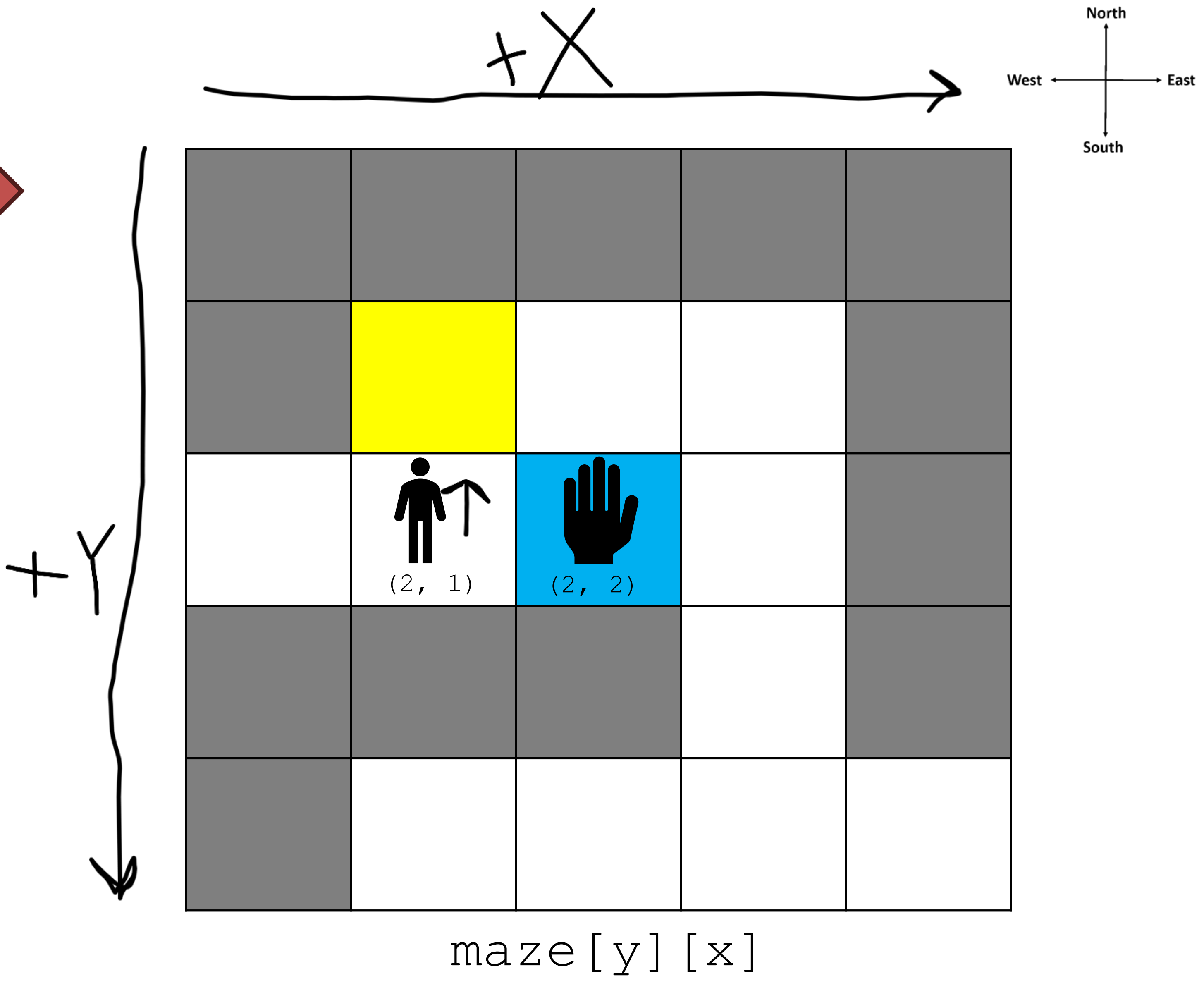


```
if(y == hand_y && hand_x > x)
        direction = "North";
}
…
```

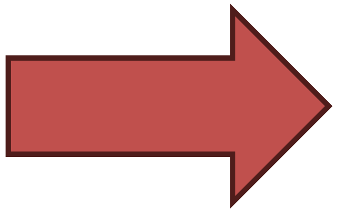
How do we detect if we can move forward?

```
char[][] maze

[[#, #, #, # ,#],
 [#, . , . , . , #],
 [. , . ,#, . , #],
 [#, #, #, . , #],
 [#, . , . , . , . ],
]
```

```
if(y == hand_y && hand_x > x)
        direction = "North";
}
…
if(maze[hand_y][hand_x] == '#' && maze[y-1][x] == '.'){


}
```



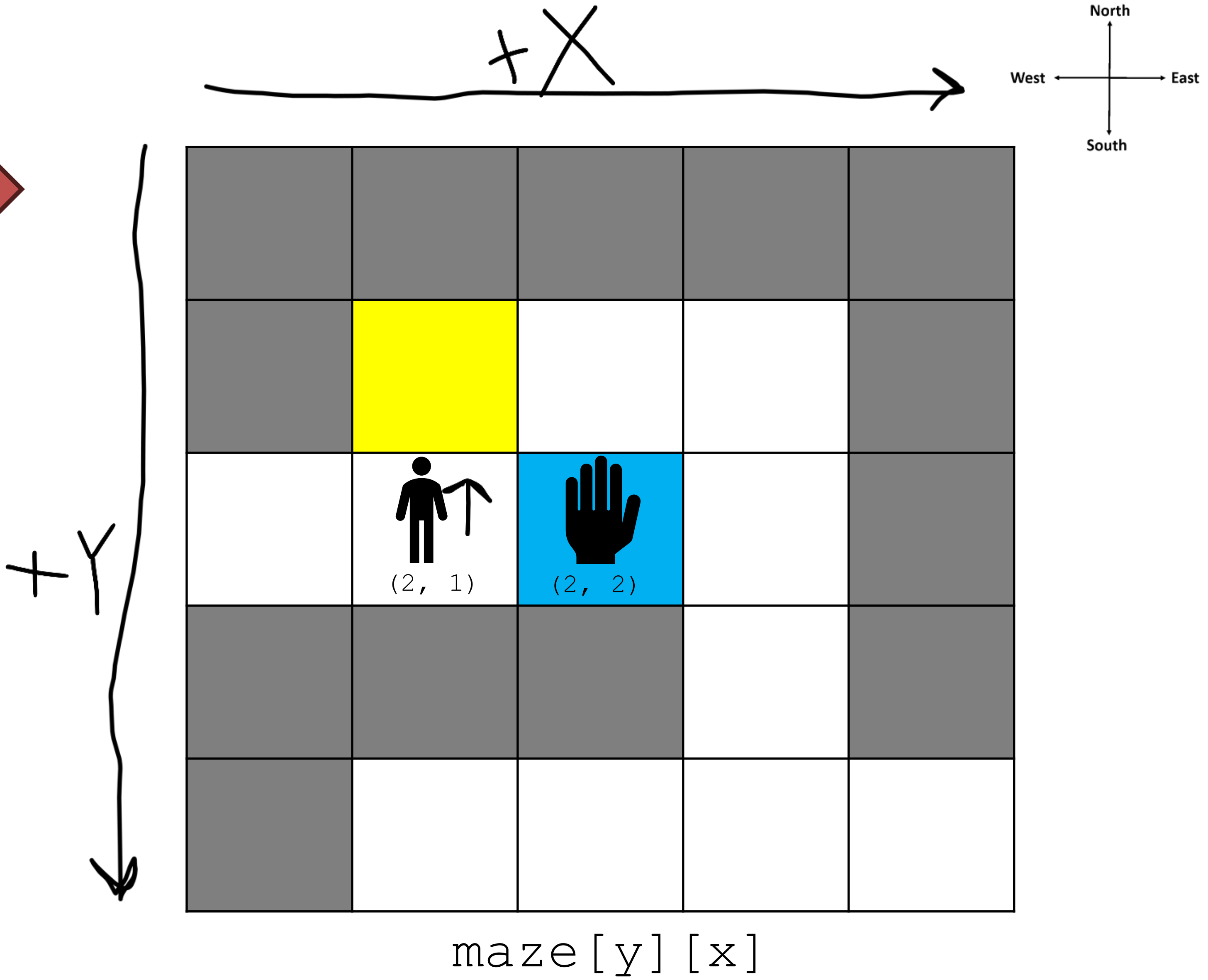
```
char[][] maze

[[#, #, #, # ,#],
 [#, ., . , . , #],
 [. , . ,#, . , #],
 [#, #, #, . , #],
 [#, . , . , . , . ],
]
```



```
if(y == hand_y && hand_x > x)
        direction = "North";
}
…
if(maze[hand_y][hand_x] == '#' && maze[y-1][x] == ‘.’){
```

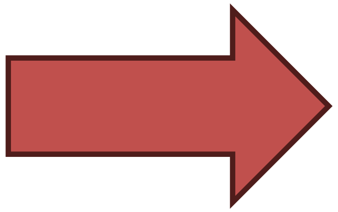
Make one move by recursively calling the method with the new values

```
}
```



```
makeMove(x, y, hand_x, hand_y)
```

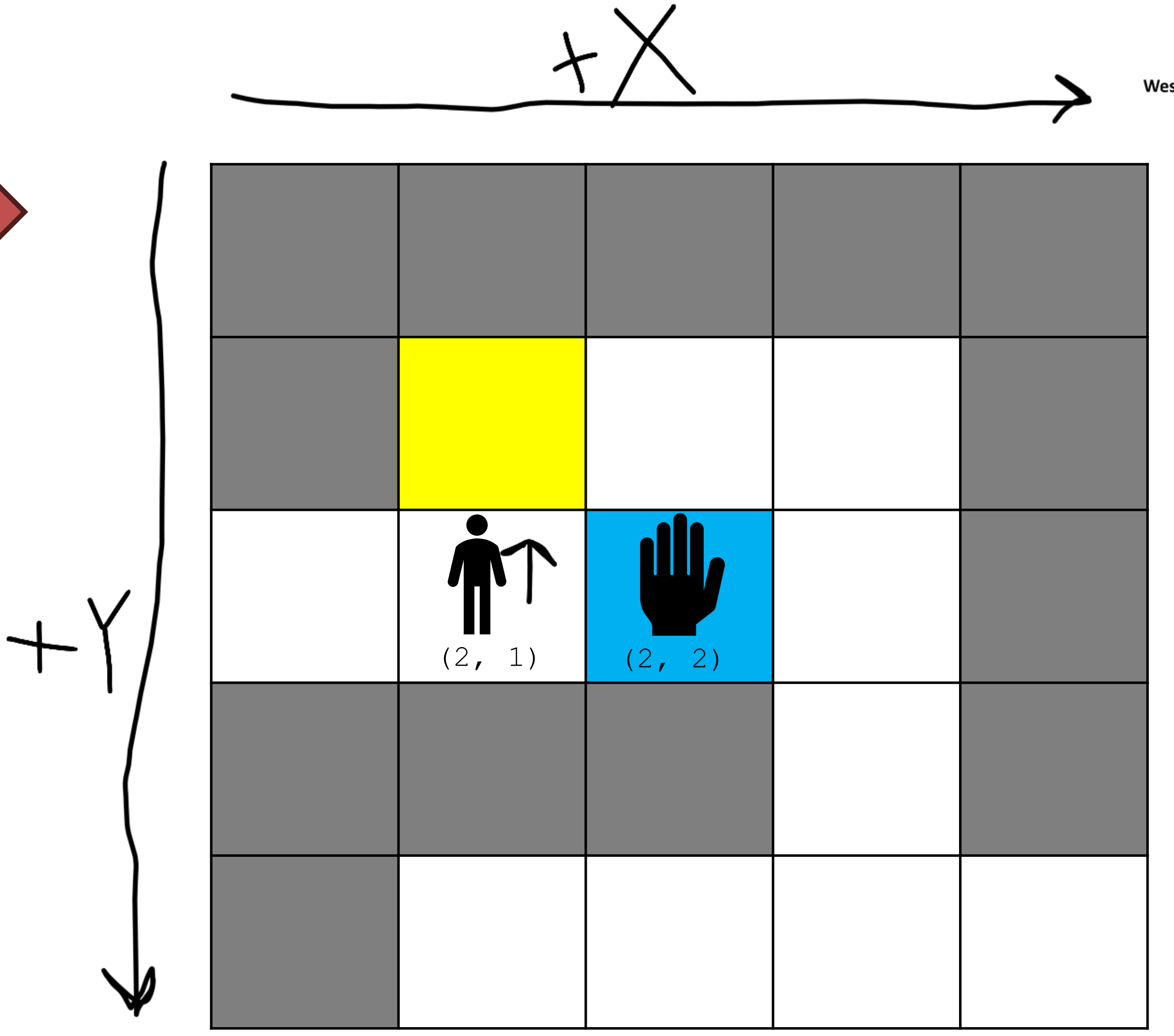
```
char[][] maze

[[#, #, #, # ,#],
 [#, ., . , . , #],
 [ . , . ,#, . , #],
 [#, #, #, . , #],
 [#, . , . , . , . ],
]
```



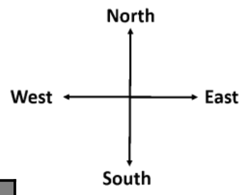
```
if(y == hand_y && hand_x > x)
        direction = "North";
}
…
if(maze[hand_y][hand_x] == '#' && maze[y-1][x] == '.'){
```

Make one move by recursively calling the method with the new values

```
}
```

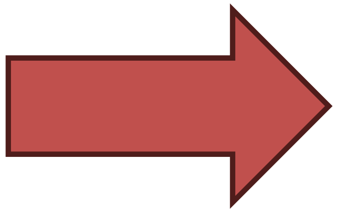


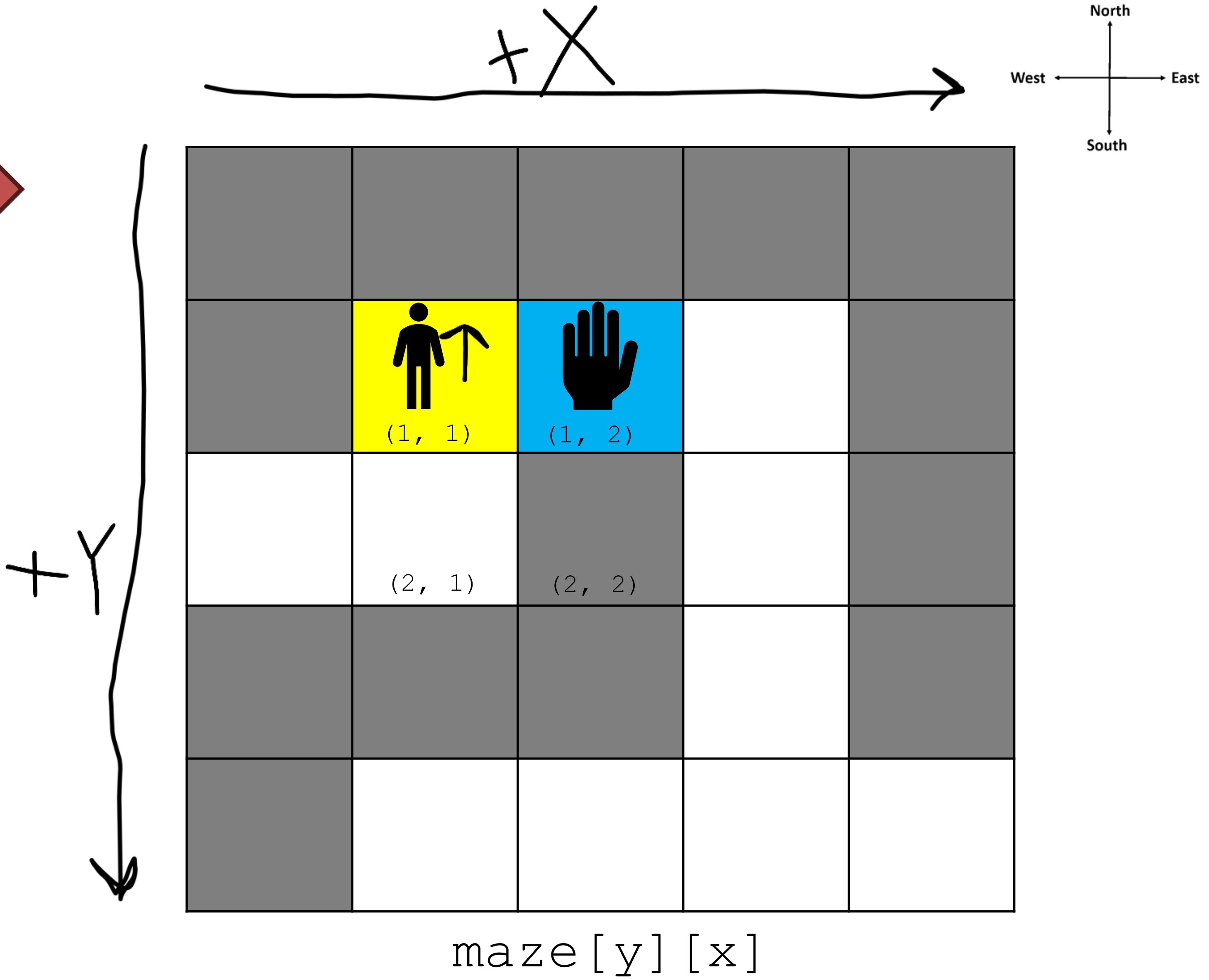
maze[y][x]



```
makeMove(x, y, hand_x, hand_y)
```

```
char[][] maze

[[#, #, #, # ,#],
 [#, . , . , . , #],
 [ . , . ,# , . , #],
 [#, #, #, . , #],
 [#, . , . , . , . ],
]
```

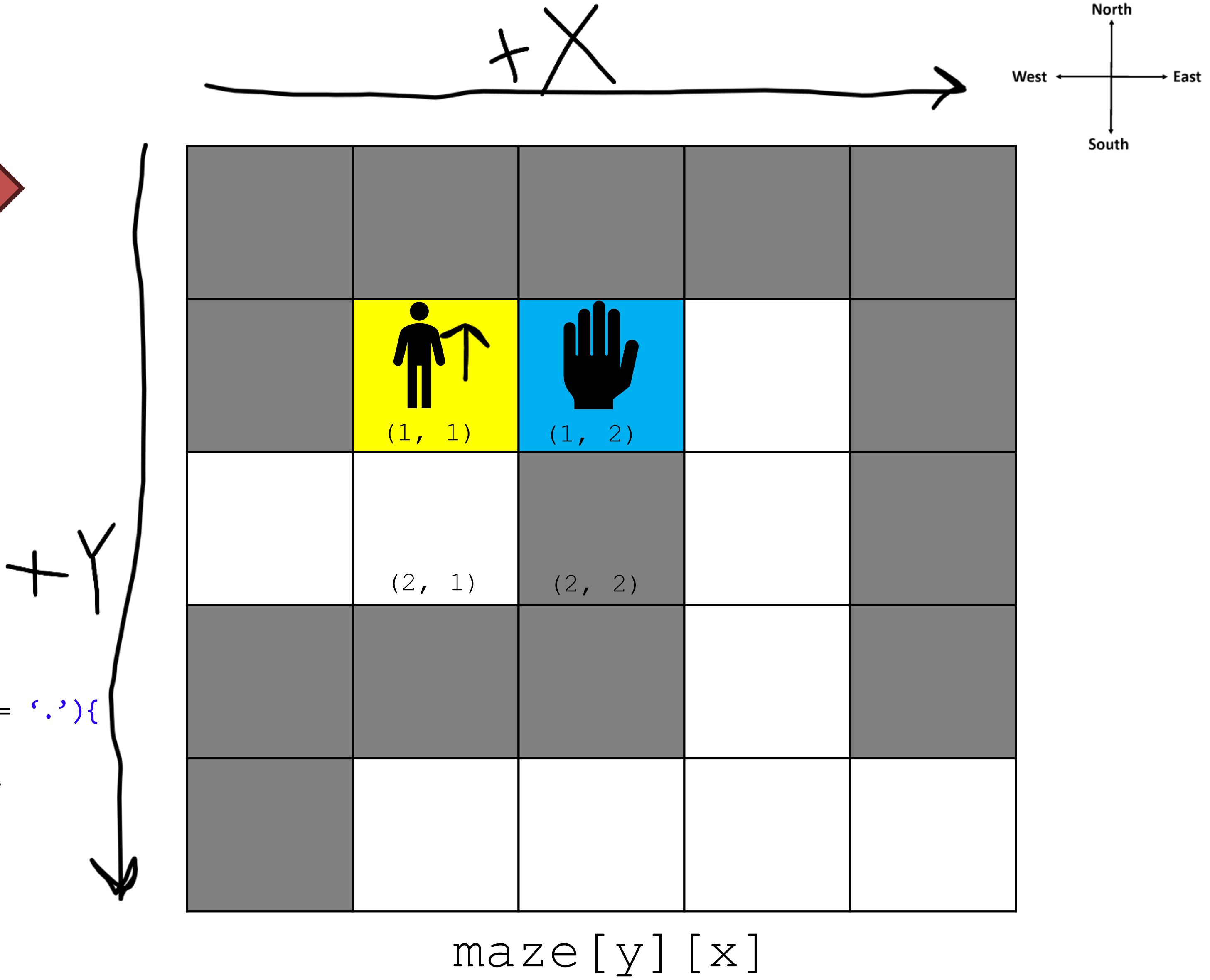




```
if(y == hand_y && hand_x > x)
        direction = "North";
}
…
if(maze[hand_y][hand_x] == '#' && maze[y-1][x] == '.'){

    makeMove(x, y-1, hand_x, hand_y-1);

}
```

```
makeMove(x, y, hand_x, hand_y)
```

```
char[][] maze

[[#, #, #, # ,#],
 [#, . , . , . , #],
 [. , . ,# , . , #],
 [#, #, #, . , #],
 [#, . , . , . , .],
]
```

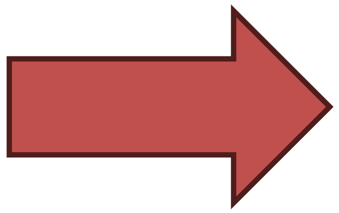


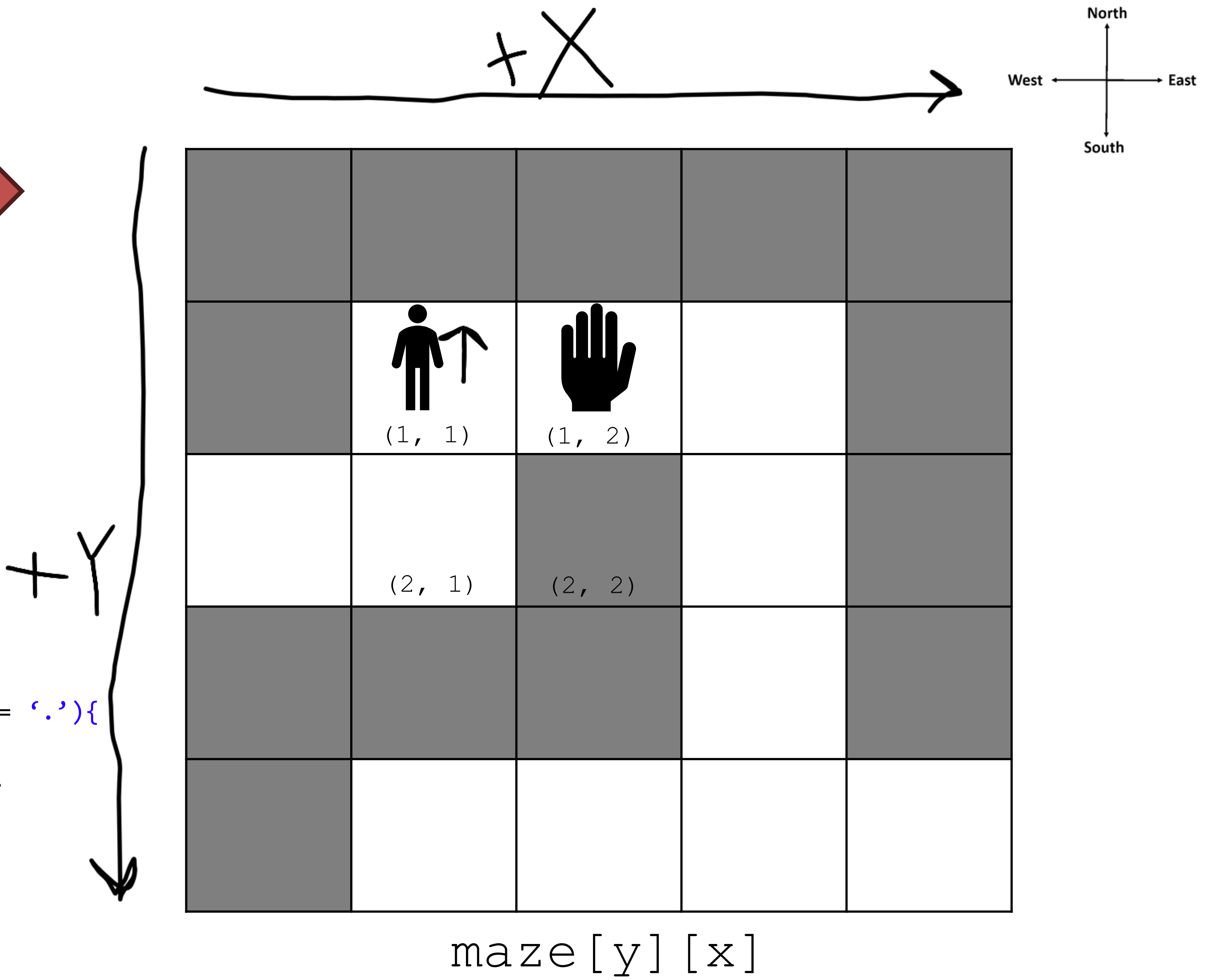
```
if(y == hand_y && hand_x > x)
        direction = "North";
}
…
if(direction.equals("North")) {
        if(maze[hand_y][hand_x] == '#' && maze[y-1][x] == '.'){

            makeMove(x, y-1, hand_x, hand_y-1);

        }
```

```
makeMove(x, y, hand_x, hand_y)
```

```
char[][] maze
```

```
[[#, #, #, # ,#],
 [#, . , . , . , #],
 [. , . ,# , . , #],
 [#, #, #, . , #],
 [#, . , . , . , .],
]
```



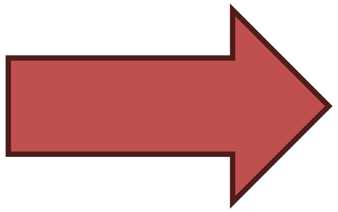


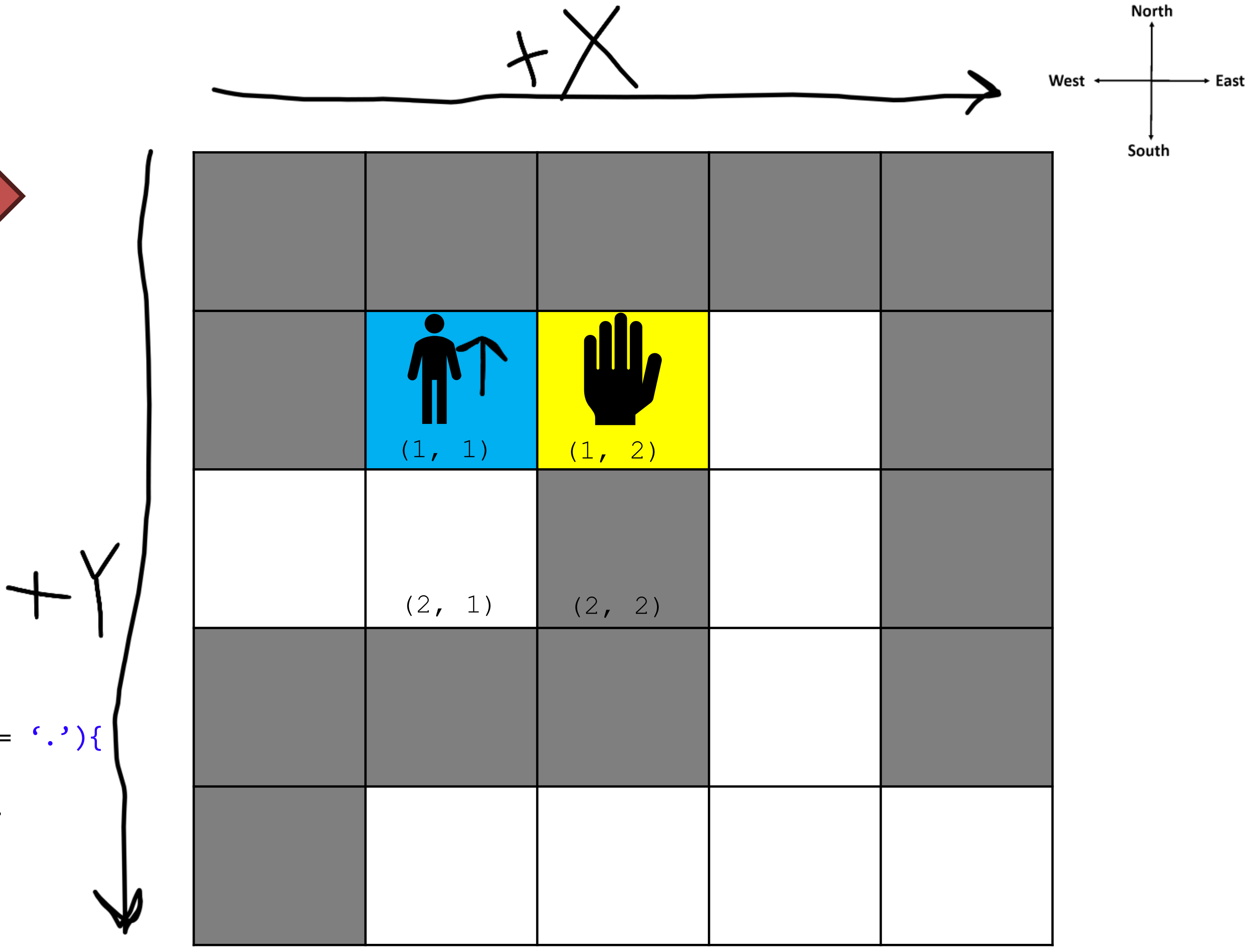
```
if(y == hand_y && hand_x > x)
        direction = "North";
}
…
if(direction.equals("North")) {
        if(maze[hand_y][hand_x] == '#' && maze[y-1][x] == '.'){

            makeMove(x, y-1, hand_x, hand_y-1);

        }
```

Turn right and move forward one spot?

```
makeMove(x, y, hand_x, hand_y)
```

```
char[][] maze

[[#, #, #, # ,#],
 [#, . , . , . , #],
 [. , . ,# , . , #],
 [#, #, #, . , #],
 [#, . , . , . , . ],
]
```

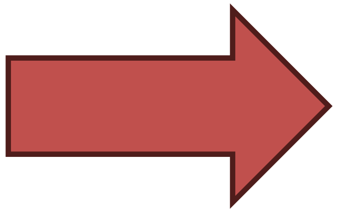


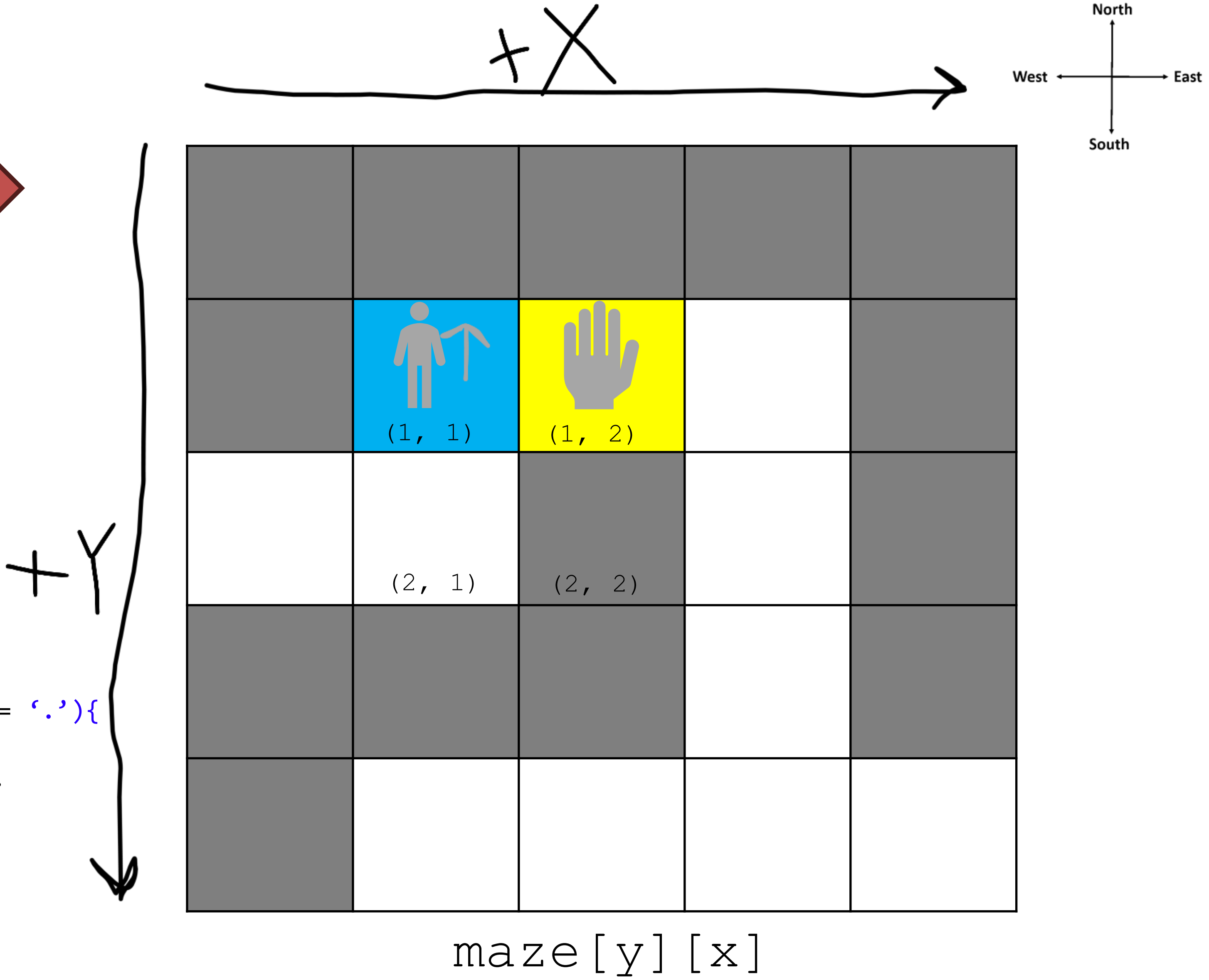


maze[y][x]

```
if(y == hand_y && hand_x > x)
        direction = "North";
}
…
if(direction.equals("North")) {
        if(maze[hand_y][hand_x] == '#' && maze[y-1][x] == '.'){

            makeMove(x, y-1, hand_x, hand_y-1);

        }
        if(maze[hand_y][hand_x] == '.'){


        }
makeMove(x, y, hand_x, hand_y)
```

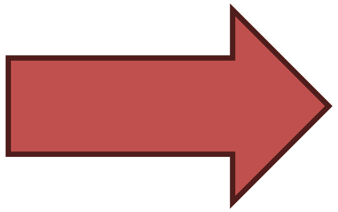
```
char[][] maze

[[#, #, #, # ,#],
 [#, . , . , . , #],
 [. , . ,# , . , #],
 [#, #, #, . , #],
 [#, . , . , . , .],
]
```



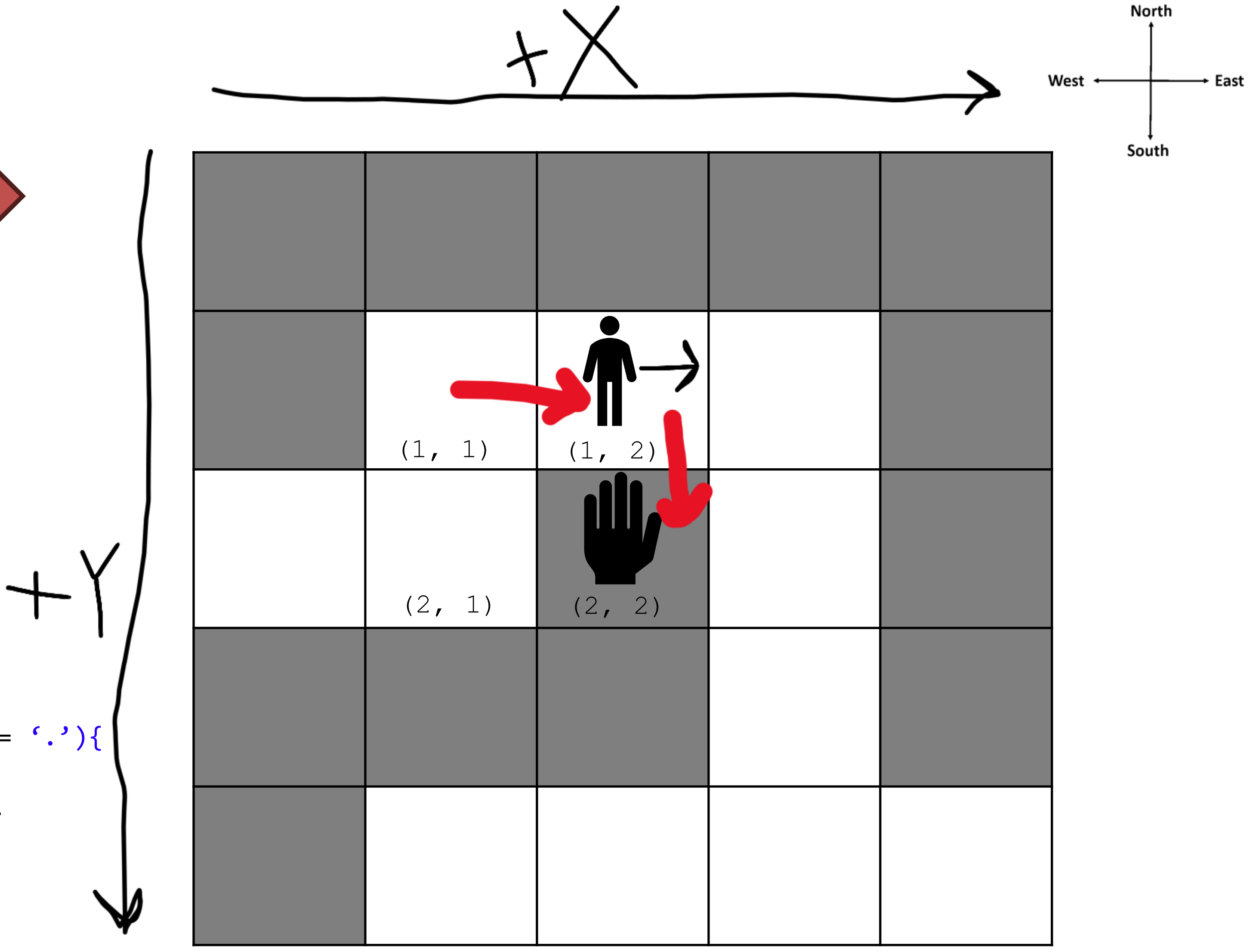


```
if(y == hand_y && hand_x > x)
        direction = "North";
}
…
if(direction.equals("North")) {
        if(maze[hand_y][hand_x] == '#' && maze[y-1][x] == '.'){

            makeMove(x, y-1, hand_x, hand_y-1);

        }
        if(maze[hand_y][hand_x] == '.'){

            makeMove(??, ??, ??, ??);

        }
makeMove(x, y, hand_x, hand_y)
```

```
char[][] maze

[[#, #, #, # ,#],
 [#, . , . , . , #],
 [. , . ,# , . , #],
 [#, #, #, . , #],
 [#, . , . , . , .],
]
```



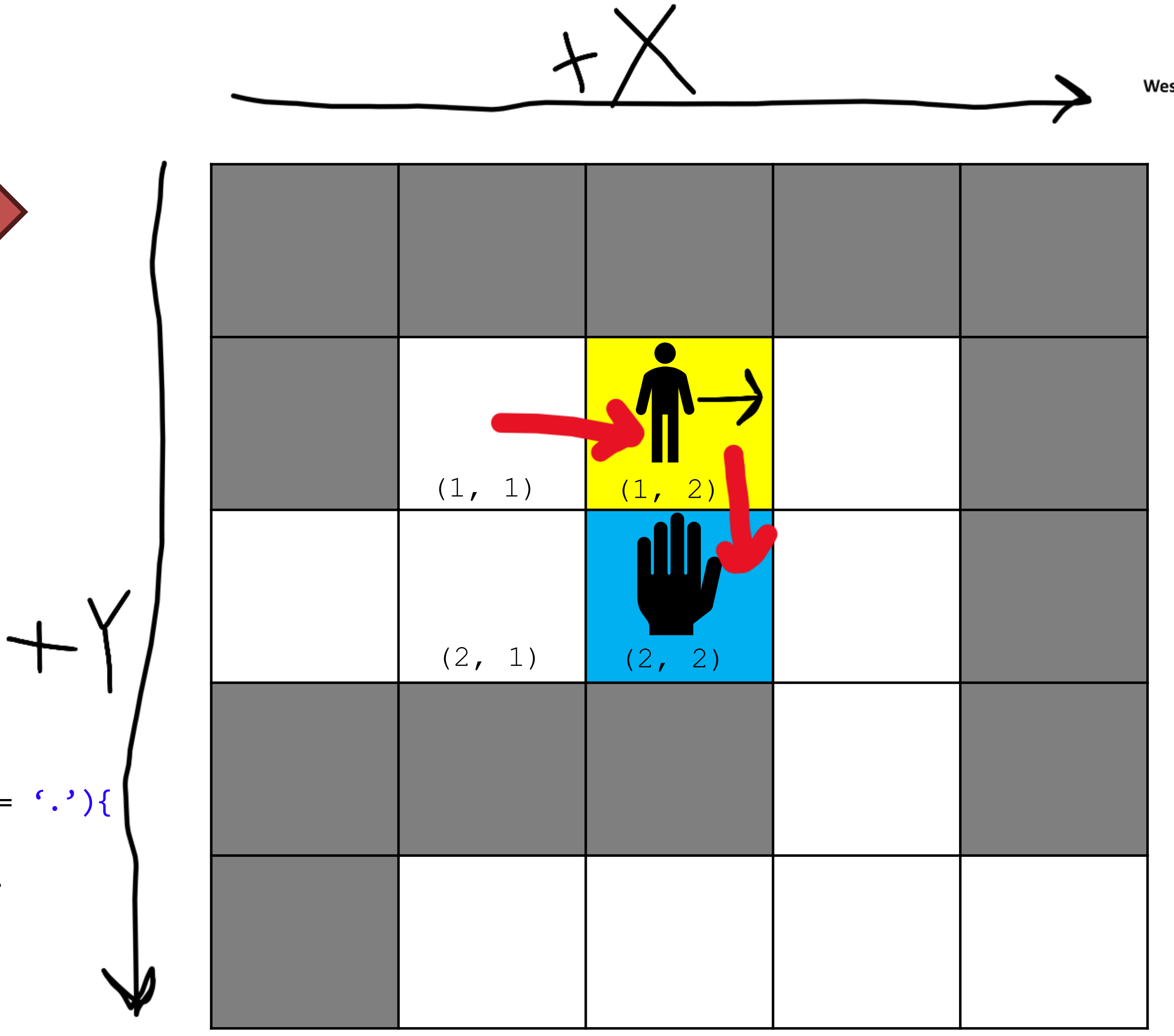
```
if(y == hand_y && hand_x > x)
        direction = "North";
}
…
if(direction.equals("North")) {
        if(maze[hand_y][hand_x] == '#' && maze[y-1][x] == ‘.’){

            makeMove(x, y-1, hand_x, hand_y-1);

        }
        if(maze[hand_y][hand_x] == ‘.’){

            makeMove(x+1, y, hand_x, hand_y+1);

        }
makeMove(x, y, hand_x, hand_y)
```



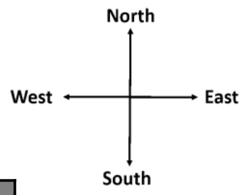
maze[y][x]

```
char[][] maze

[[#, #, #, # ,#],
 [#, . , . , . , #],
 [. , . ,# , . , #],
 [#, #, #, . , #],
 [#, . , . , . , .],
]
```

```
if(y == hand_y && hand_x > x)
        direction = "North";
}
…
if(direction.equals("North")) {
        if(maze[hand_y][hand_x] == '#' && maze[y-1][x] == '.'){

            makeMove(x, y-1, hand_x, hand_y-1);

        }
        if(maze[hand_y][hand_x] == '.'){

            makeMove(x+1, y, hand_x, hand_y+1);

        }
makeMove(x, y, hand_x, hand_y)
```

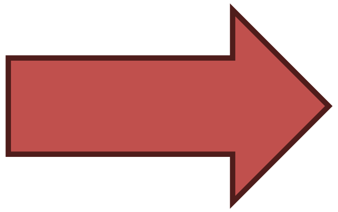


maze[y][x]

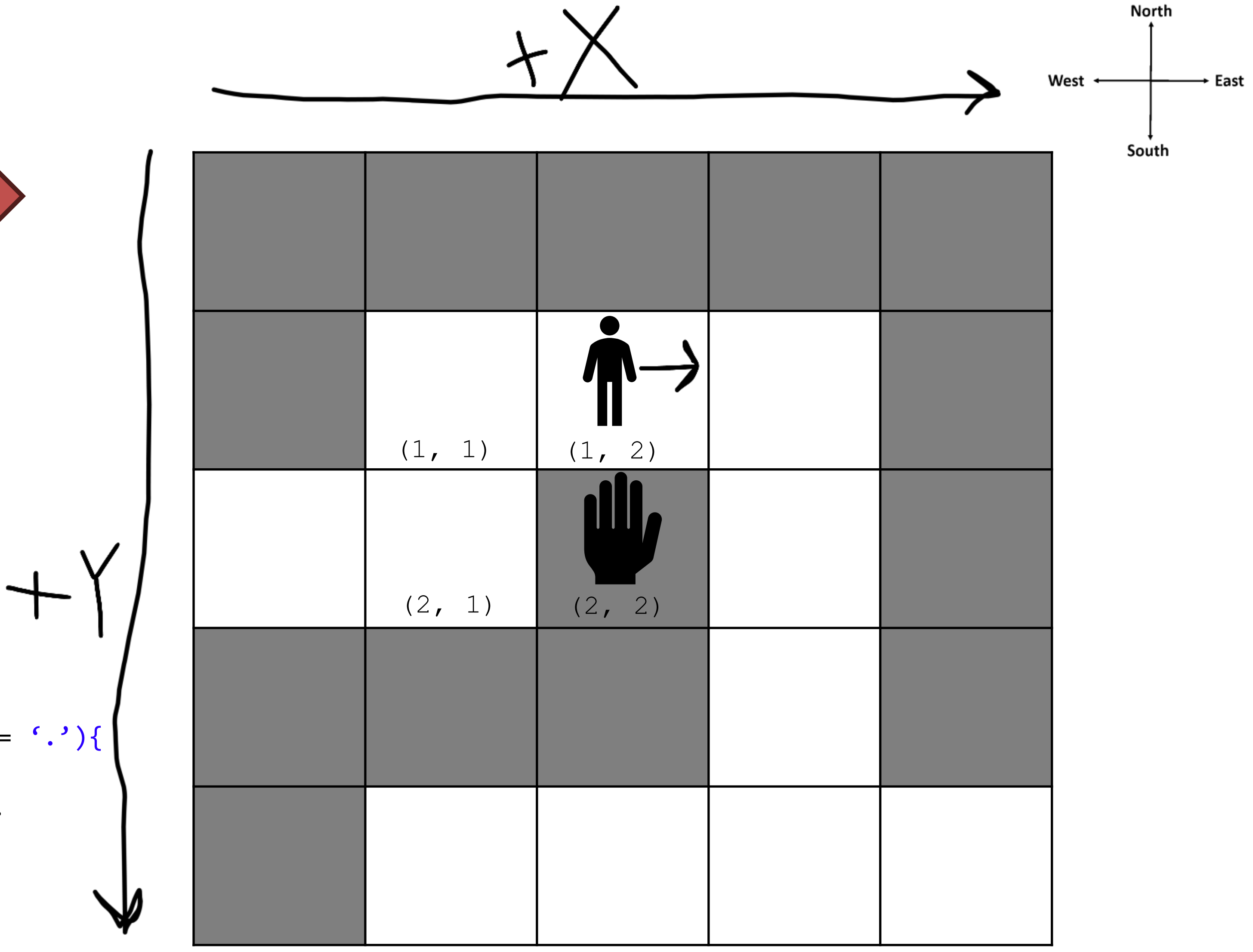


char[][] maze

```
[[#, #, #, # ,#],
 [#, . , . , . , #],
 [ . , . ,# , . , #],
 [#, #, #, . , #],
 [#, . , . , . , . ],
]
```



```
if(y == hand_y && hand_x > x)
        direction = "North";
}
…
if(direction.equals("North")) {
      if(maze[hand_y][hand_x] == '#' && maze[y-1][x] == '.'){

         makeMove(x, y-1, hand_x, hand_y-1);

      }
      if(maze[hand_y][hand_x] == '.'){

         makeMove(x+1, y, hand_x, hand_y+1);

      }
makeMove(x, y, hand_x, hand_y)
```

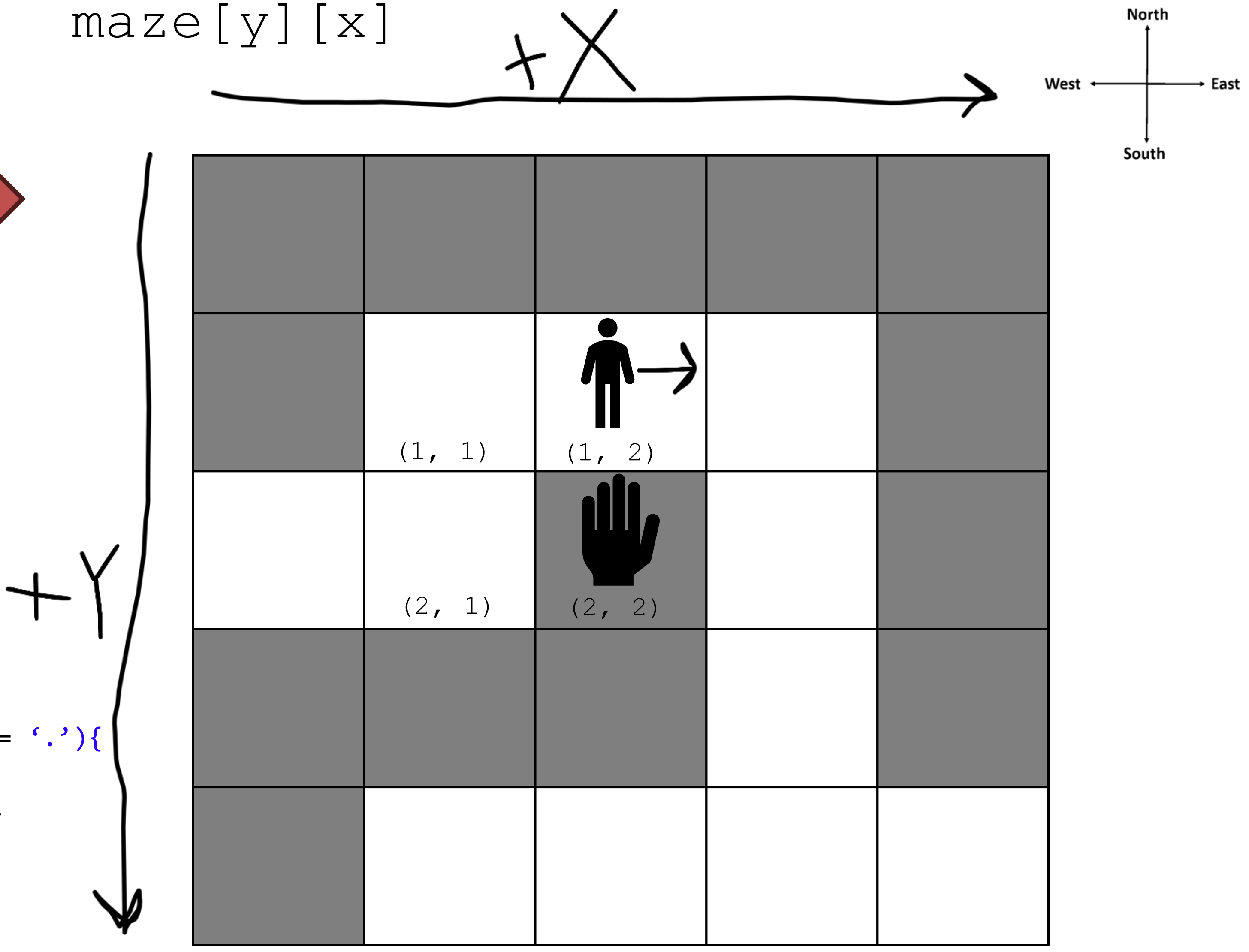


maze[y][x]

char[][] maze

```
[[#, #, #, # ,#],
 [#, . , . , . , #],
 [. , . ,# , . , #],
 [#, #, #, . , #],
 [#, . , . , . , . ],
]
```

```
if(y == hand_y && hand_x > x)
        direction = "North";
}
…
if(direction.equals("North")) {
        if(maze[hand_y][hand_x] == '#' && maze[y-1][x] == '.'){

            makeMove(x, y-1, hand_x, hand_y-1);

        }
        if(maze[hand_y][hand_x] == '.'){

            makeMove(x+1, y, hand_x, hand_y+1);

        }
makeMove(x, y, hand_x, hand_y)
```



1. Turn right
2. Go forward
3. Turn left

```
char[][] maze

[[#, #, #, # ,#],
 [#, . , . , . , #],
 [. , . ,# , . , #],
 [#, #, #, . , #],
 [#, . , . , . , .],
]
```

```
if(y == hand_y && hand_x > x)
        direction = "North";
}
…
if(direction.equals("North")) {
```

Right

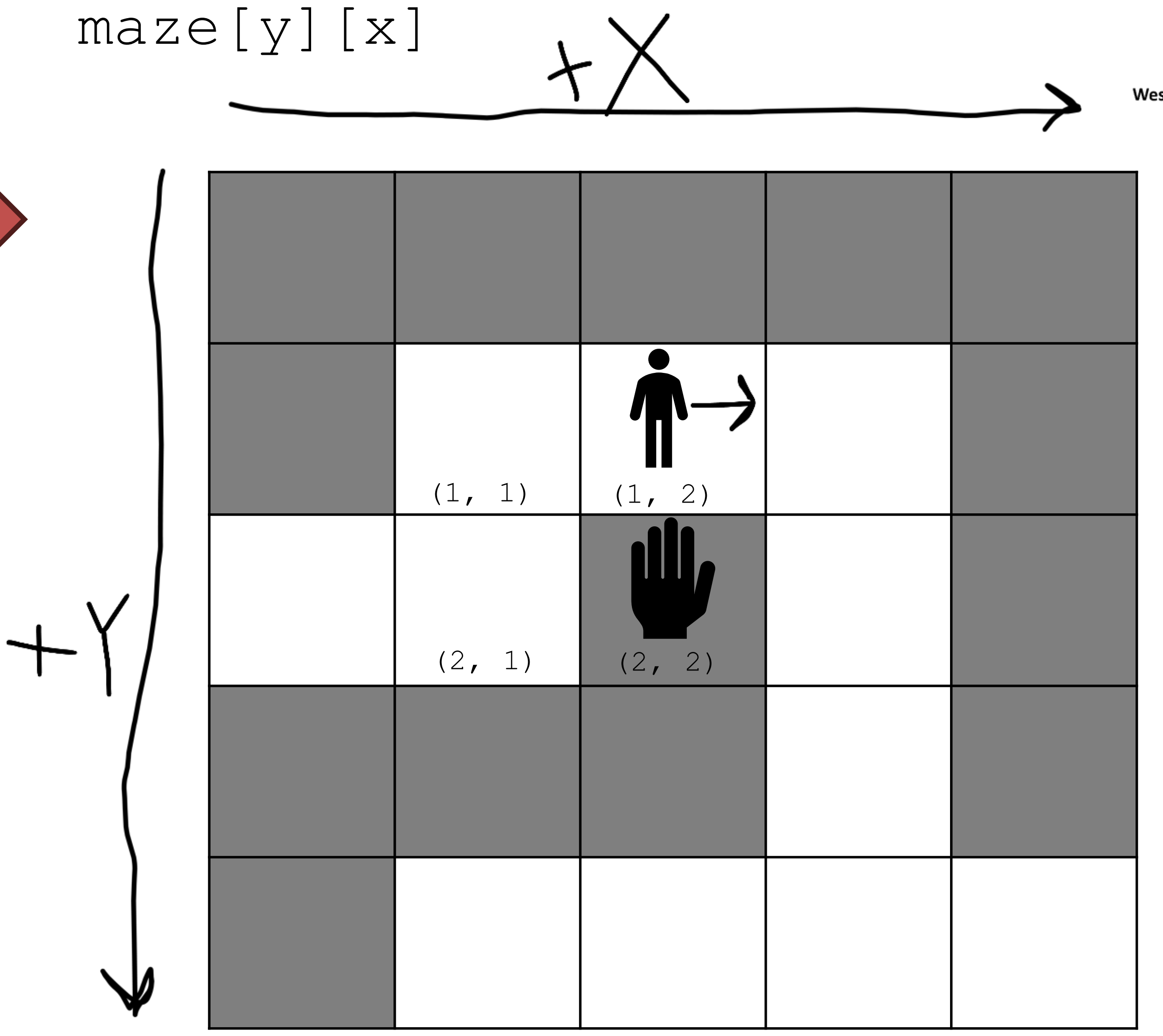
```
        if(maze[hand_y][hand_x] == '.'){
           makeMove(x+1, y, hand_x, hand_y+1);

        }
```

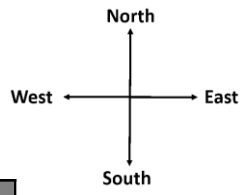
Forward

```
        if(maze[hand_y][hand_x] == '#' && maze[y-1][x] == '.'){

           makeMove(x, y-1, hand_x, hand_y-1);

        }
```

Left

```
// Turn left
```





1. Turn right
2. Go forward
3. Turn left

MONTANA STATE UNIVERSITY

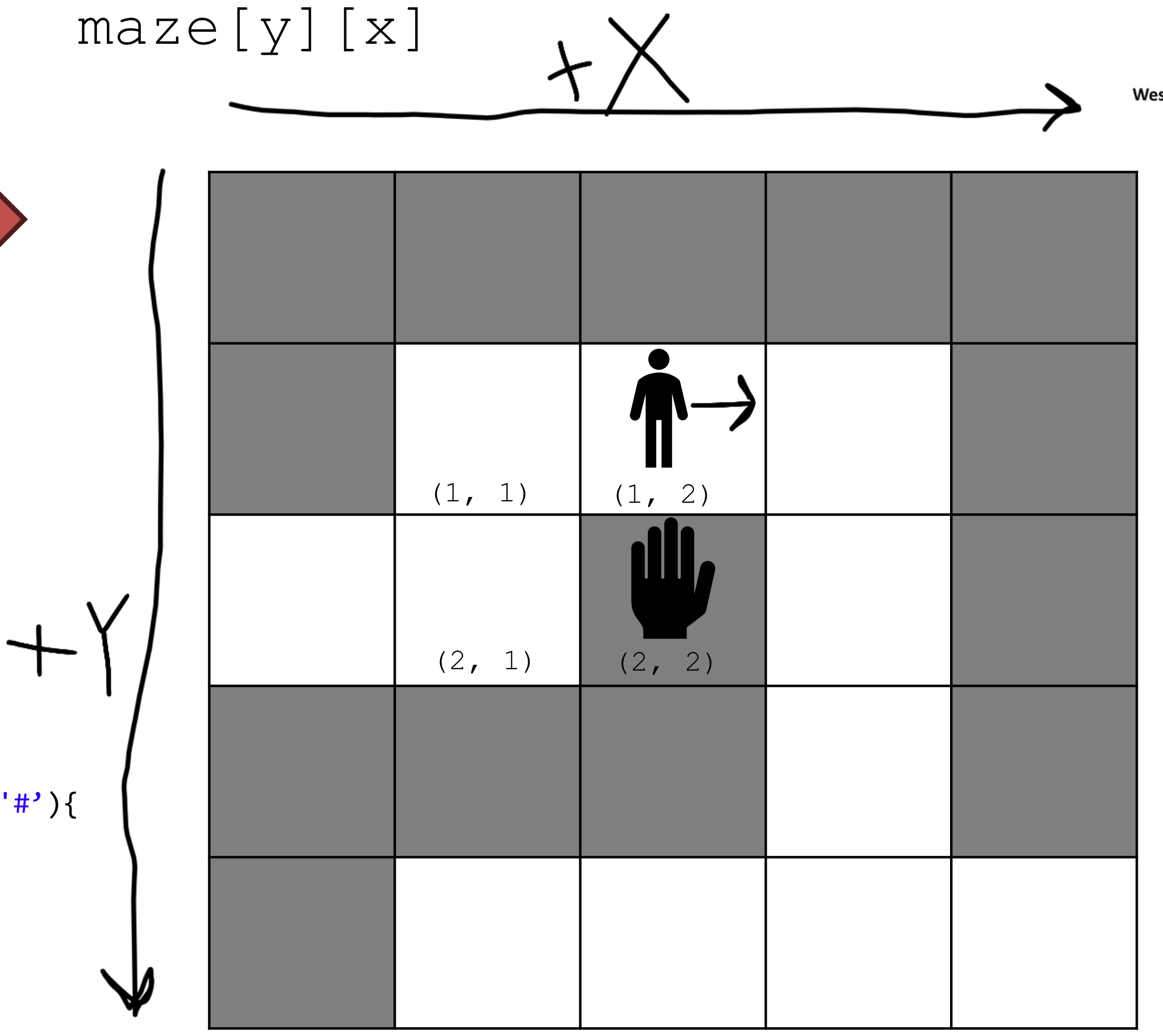
```
char[][] maze

[[#, #, #, # ,#],
 [#, . , . , . , #],
 [. , . ,# , . , #],
 [#, #, #, . , #],
 [#, . , . , . , .],
]
```

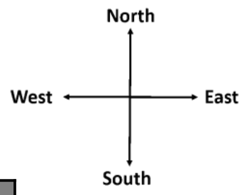
```
if(y == hand_y && hand_x > x)
        direction = "North";
}
…
if(direction.equals("North")) {
        if(maze[hand_y][hand_x] == '.' && maze[y-1][x]=='#'){
          makeMove(x+1, y, hand_x, hand_y+1);

        }

        if(maze[hand_y][hand_x] == '#' && maze[y-1][x] == '.'){

          makeMove(x, y-1, hand_x, hand_y-1);

        }
        // Turn left
```

Right

Forward

Left





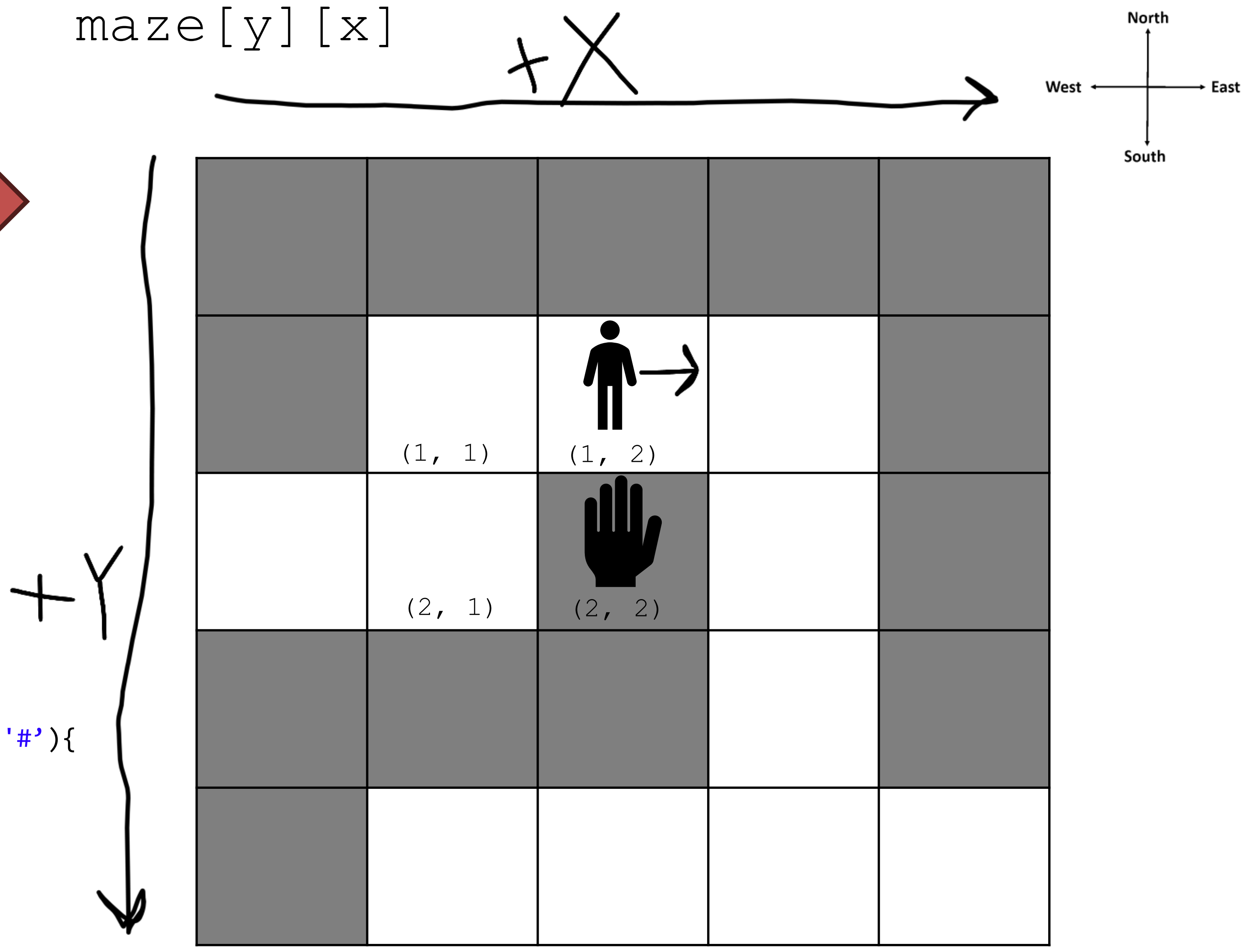
You will have need if statements for North, East, South, and West

Lots of if statements ☺

char[][] maze

```
[[#, #, #, # ,#],
 [#, . , . , . , #],
 [. , . ,# , . , #],
 [#, #, #, . , #],
 [#, . , . , . , .],
]
```

maze[y][x]



```
if(y == hand_y && hand_x > x)
        direction = "North";
}
…
if(direction.equals("North")) {
        if(maze[hand_y][                    aze[y-1][x]=='#'){
           make                              +1);

        }

        if(ma                                == '.'){
           make                              -1);

        }
// Turn left
```

Right

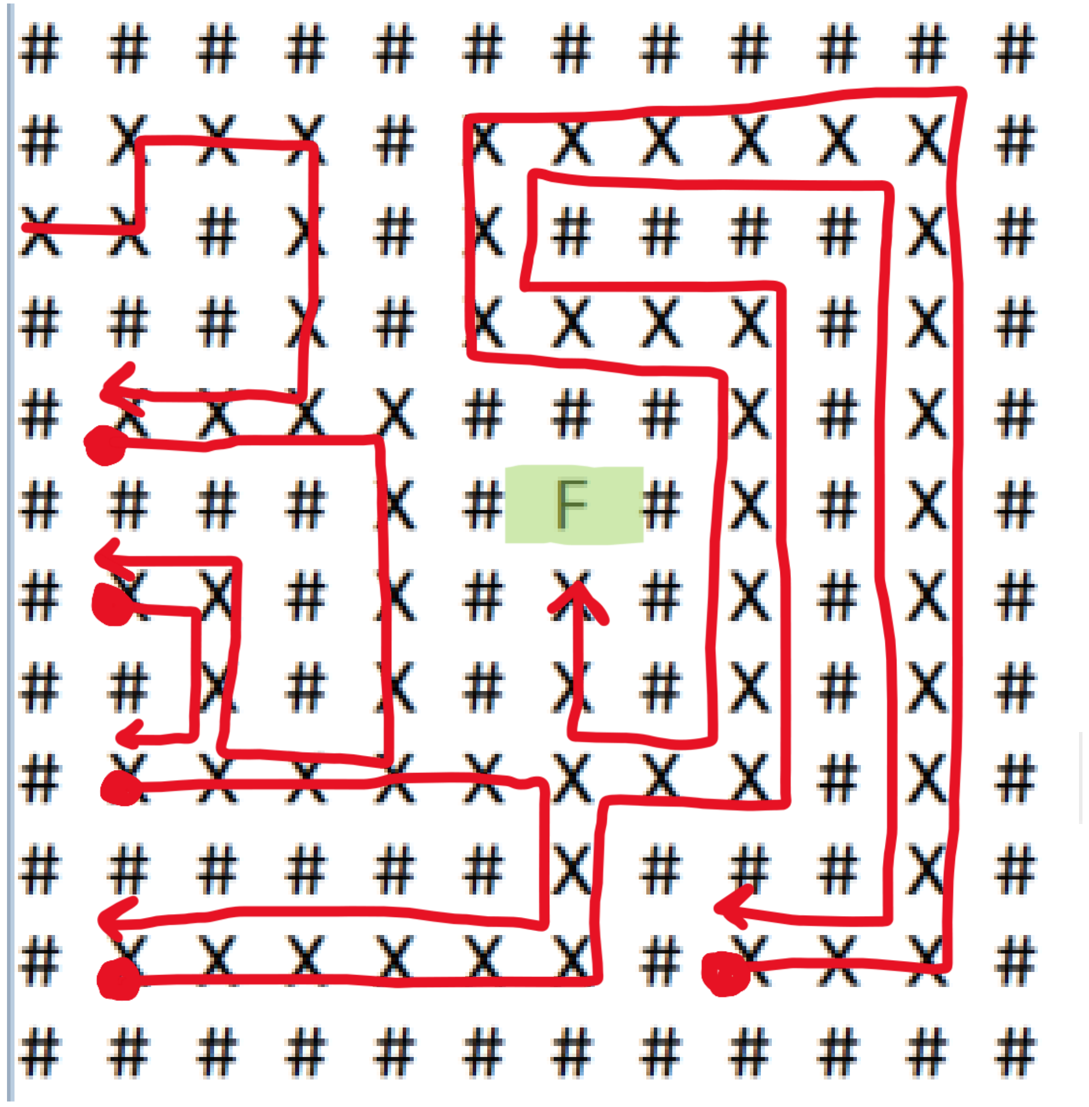
Forward

Left

This code is technically not complete, you will need to add some more code here (backtracking)

You will have need if statements for North, East, South, and West

Lots of if statements ☺



```
# # # # # # # # # # # #
# X X X # X X X X X X #
X X # X # X # # # # X #
# # # X # X X X X # X #
# X X X X # # # X # X #
# # # # X # F # X # X #
# X X # X # X # X # X #
# # X # X # X # X # X #
# X X X X X X X X # X #
# # # # # # X # # # X #
# X X X X X X # X X X #
# # # # # # # # # # # #
```

● = Backtracking path

Running Time of Sorting Algorithms

	Brief Description	Running Time
Bubble Sort	???	???
Selection Sort	???	???
Merge Sort	???	???
Quick Sort	???	???

```
public int[] selectionSort(int[] array) {
        int n = array.length;
        for(int i = 0; i < n -1; i++) {
                int min_index_so_far = i;
                for (int j = i + 1; j < n; j++) {
                        if(array[j] < array[min_index_so_far]) {
                                min_index_so_far = j;
                        }
                }
                int temp = array[i];
                array[i] = array[min_index_so_far];
                array[min_index_so_far] = temp;
        }
        return array;
}
```

You will not be tested about today’s sorting algorithms.

Insertion Sort

We divide our array into two sections. A sorted section, and an unsorted section. We iterate through the array, and for each iteration, we move one element from the unsorted section to the sorted section

38	27	43	3	9	82	10	14

Insertion Sort

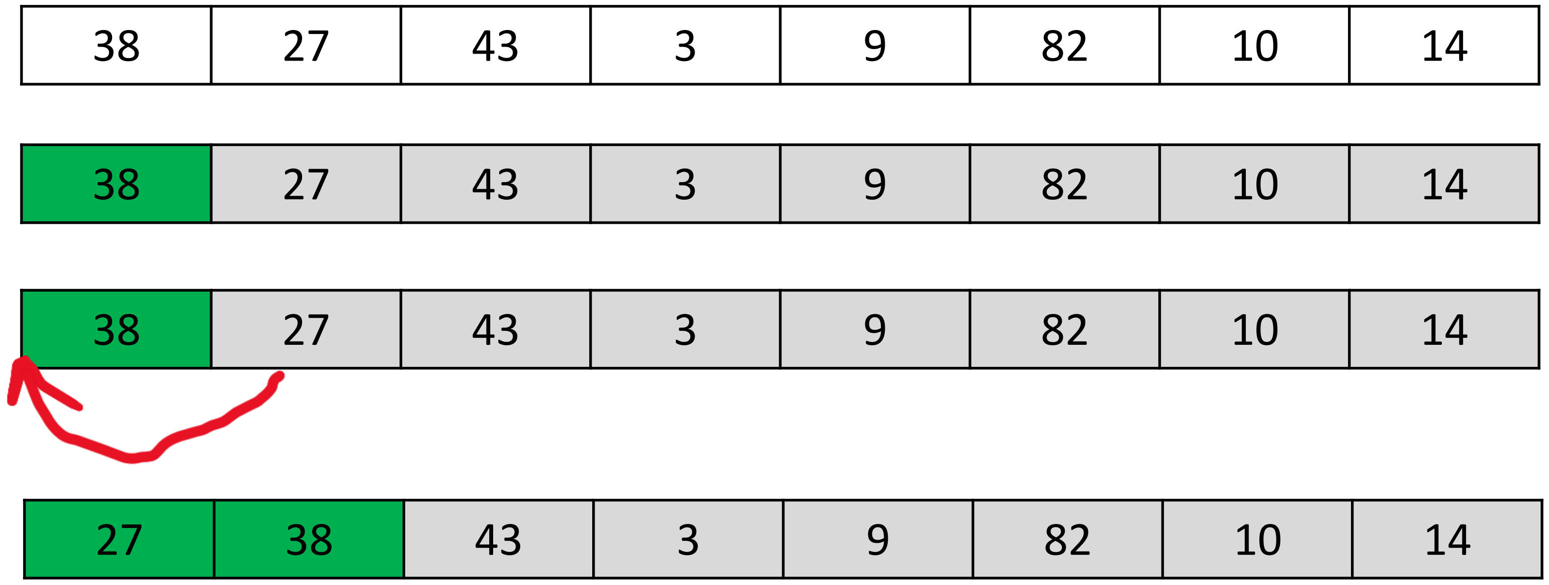
We divide our array into two sections. A sorted section, and an unsorted section. We iterate through the array, and for each iteration, we move one element from the unsorted section to the sorted section

38	27	43	3	9	82	10	14

🟩 38	27	43	3	9	82	10	14

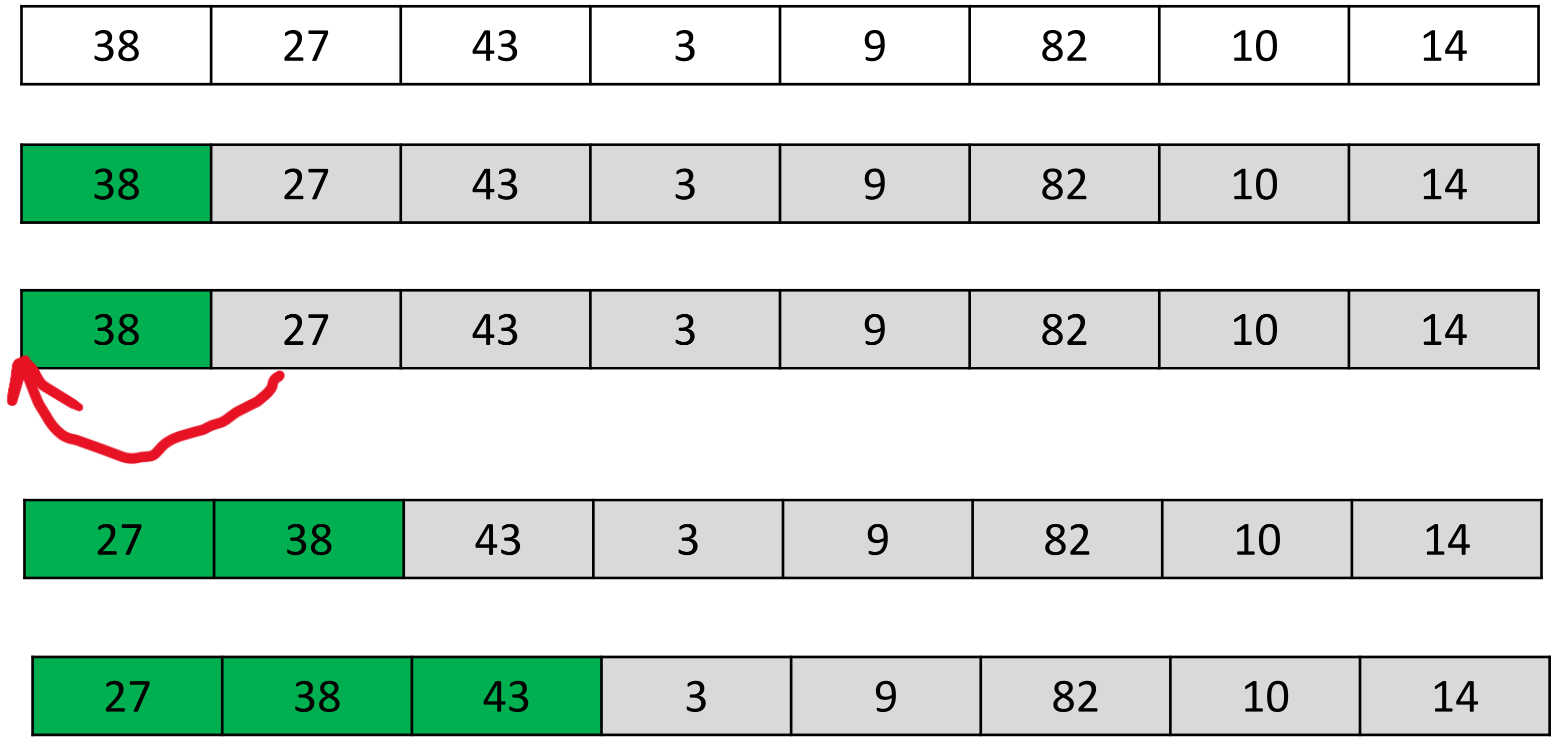
Insertion Sort

We divide our array into two sections. A sorted section, and an unsorted section. We iterate through the array, and for each iteration, we move one element from the unsorted section to the sorted section



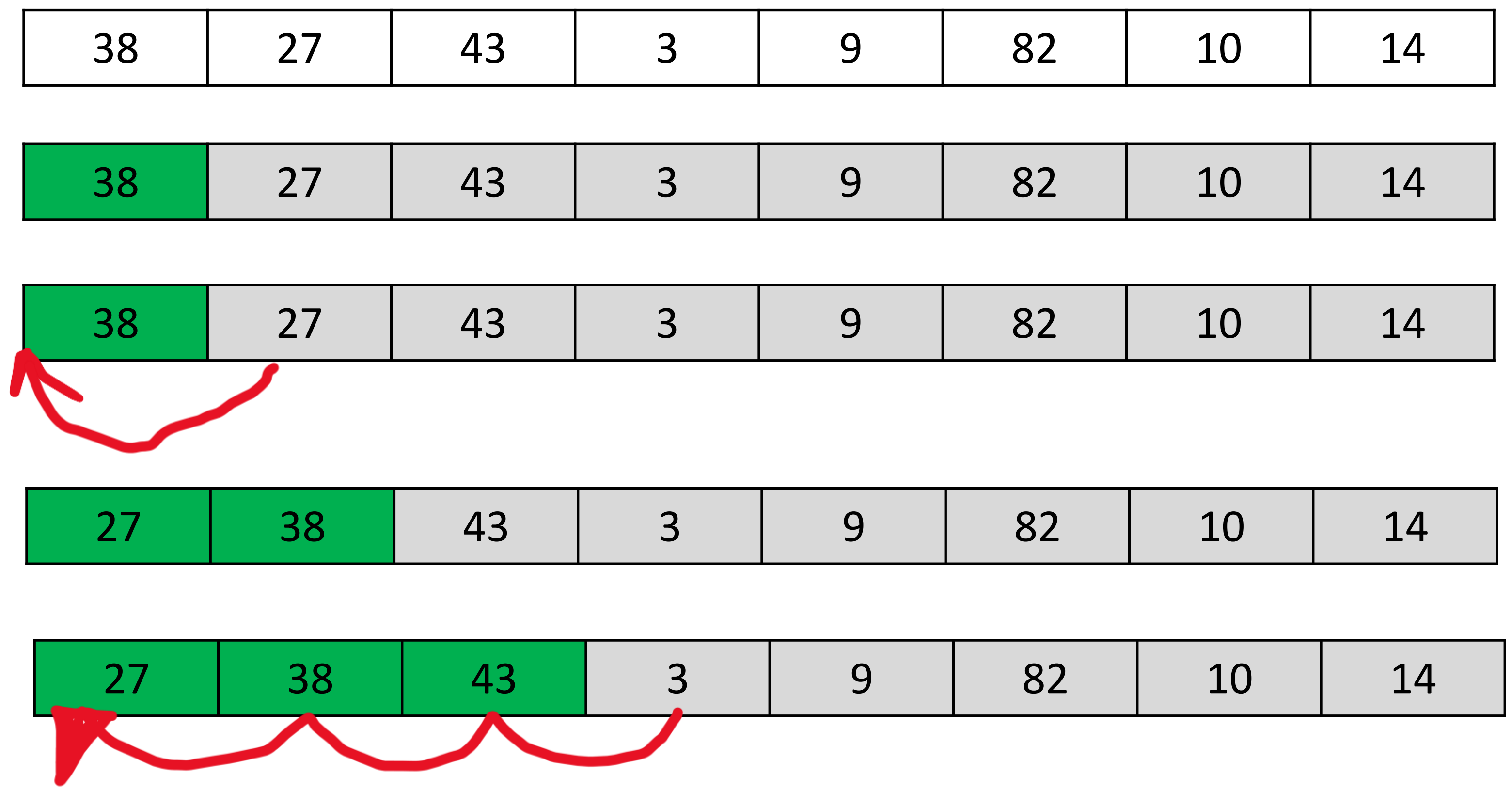
Insertion Sort

We divide our array into two sections. A sorted section, and an unsorted section. We iterate through the array, and for each iteration, we move one element from the unsorted section to the sorted section



Insertion Sort

We divide our array into two sections. A sorted section, and an unsorted section. We iterate through the array, and for each iteration, we move one element from the unsorted section to the sorted section



38	27	43	3	9	82	10	14

🟩 38	27	43	3	9	82	10	14

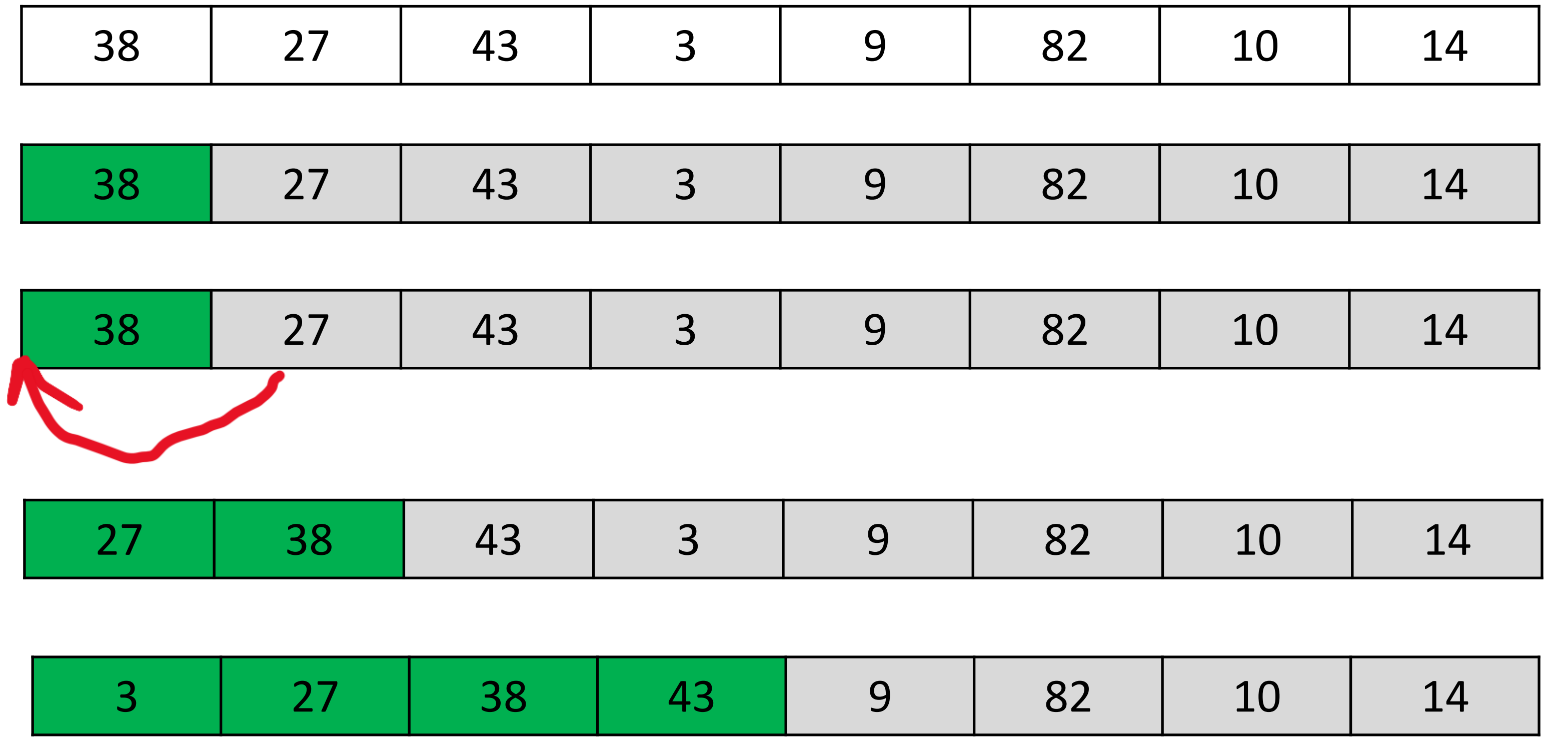
🟩 38	27	43	3	9	82	10	14

🟩 27	🟩 38	43	3	9	82	10	14

🟩 27	🟩 38	🟩 43	3	9	82	10	14

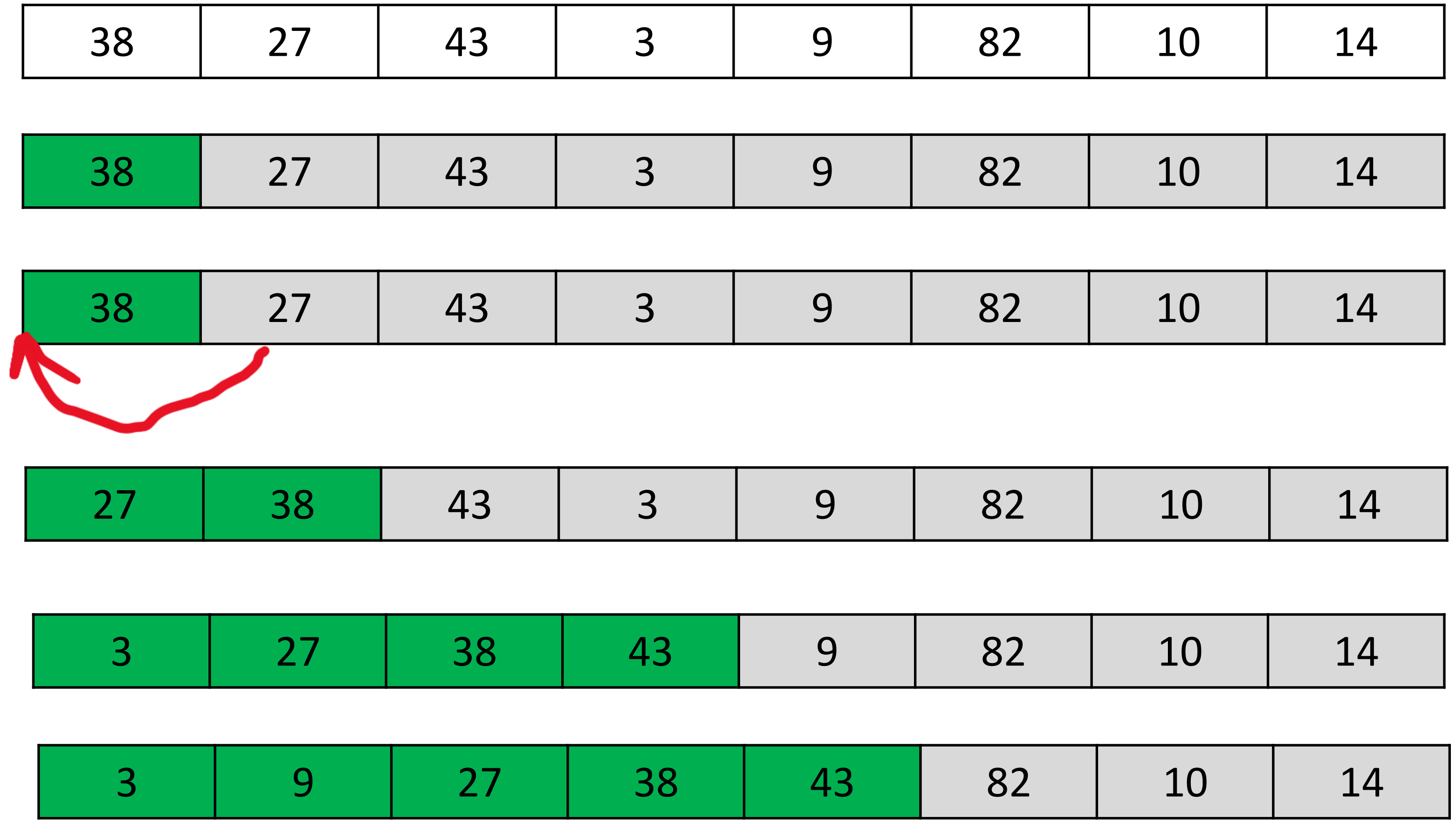
Insertion Sort

We divide our array into two sections. A sorted section, and an unsorted section. We iterate through the array, and for each iteration, we move one element from the unsorted section to the sorted section



Insertion Sort

We divide our array into two sections. A sorted section, and an unsorted section. We iterate through the array, and for each iteration, we move one element from the unsorted section to the sorted section



38	27	43	3	9	82	10	14

🟩 38	27	43	3	9	82	10	14

🟩 38	27	43	3	9	82	10	14

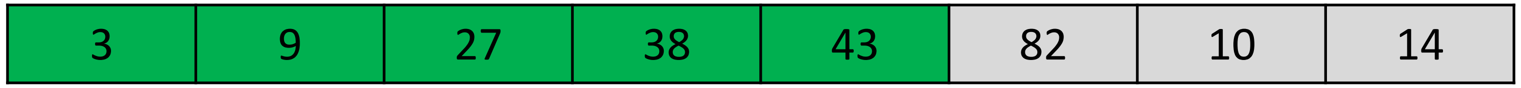
🟩 27	🟩 38	43	3	9	82	10	14

🟩 3	🟩 27	🟩 38	🟩 43	9	82	10	14

🟩 3	🟩 9	🟩 27	🟩 38	🟩 43	82	10	14

Insertion Sort

We divide our array into two sections. A sorted section, and an unsorted section. We iterate through the array, and for each iteration, we move one element from the unsorted section to the sorted section



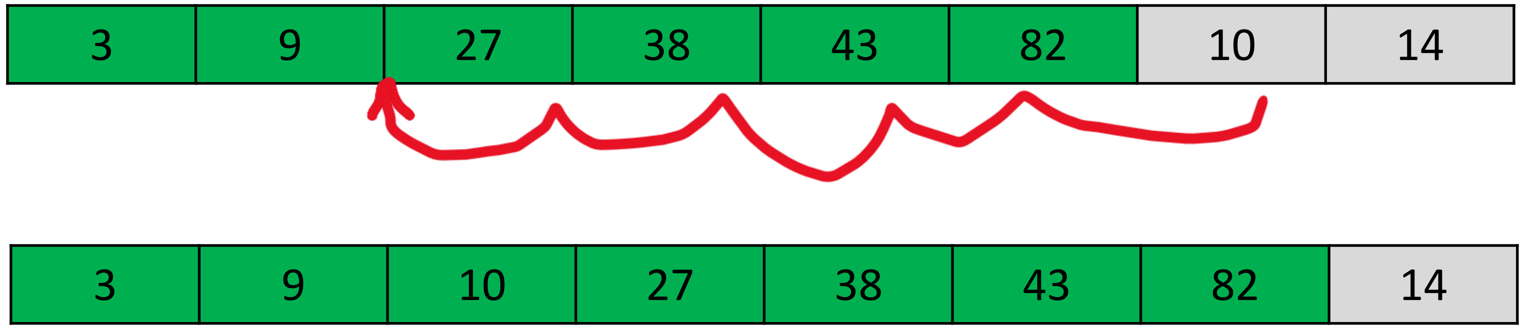
Insertion Sort

We divide our array into two sections. A sorted section, and an unsorted section. We iterate through the array, and for each iteration, we move one element from the unsorted section to the sorted section

3	9	27	38	43	82	10	14

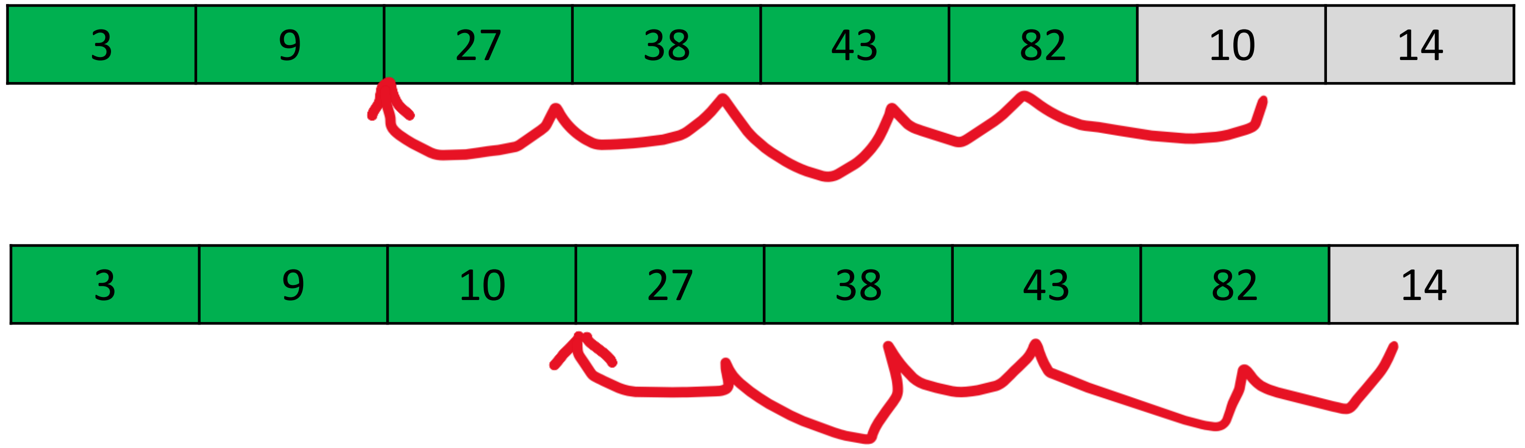
Insertion Sort

We divide our array into two sections. A sorted section, and an unsorted section. We iterate through the array, and for each iteration, we move one element from the unsorted section to the sorted section



Insertion Sort

We divide our array into two sections. A sorted section, and an unsorted section. We iterate through the array, and for each iteration, we move one element from the unsorted section to the sorted section

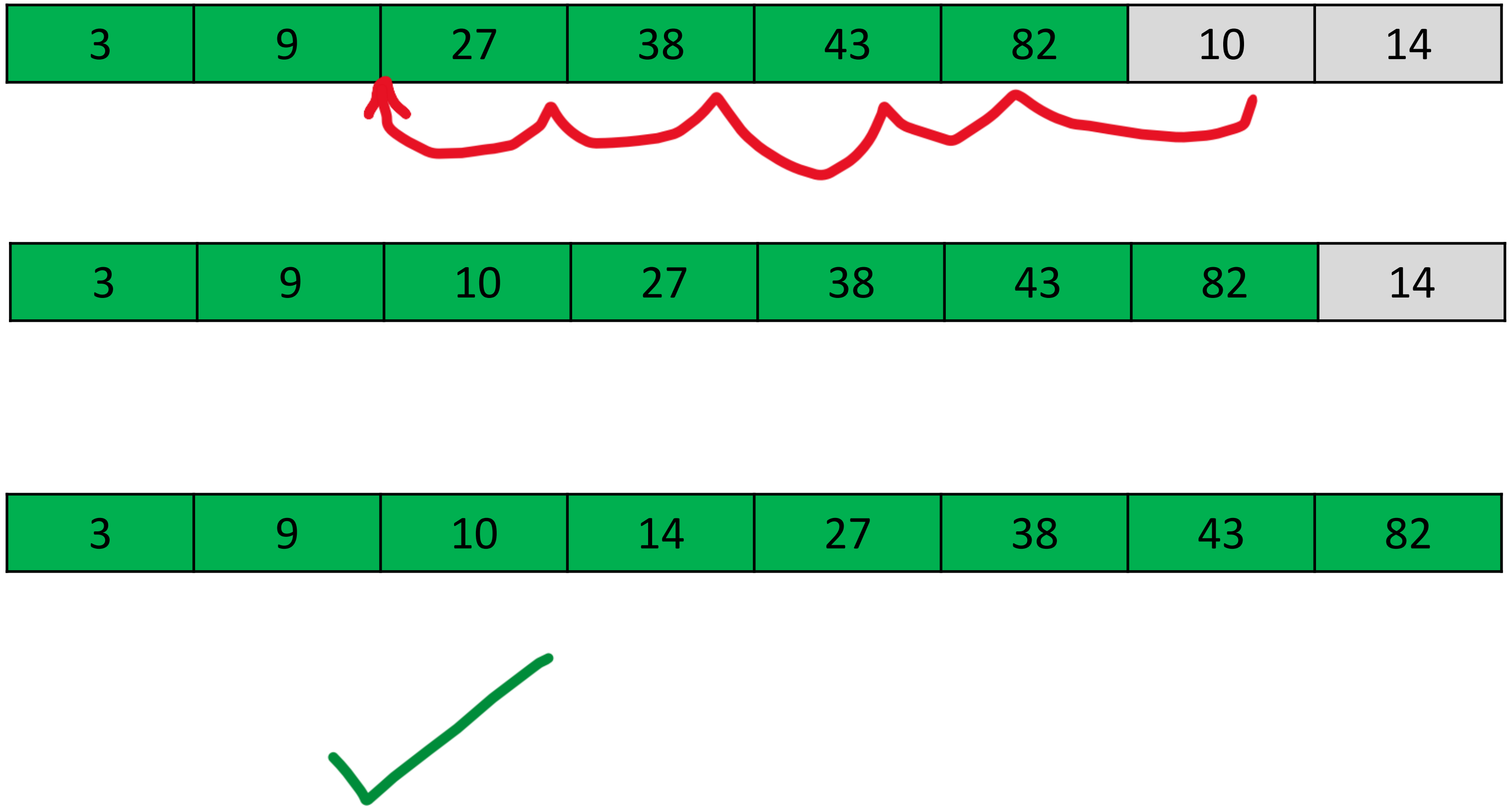


3	9	27	38	43	82	10	14

3	9	10	27	38	43	82	14

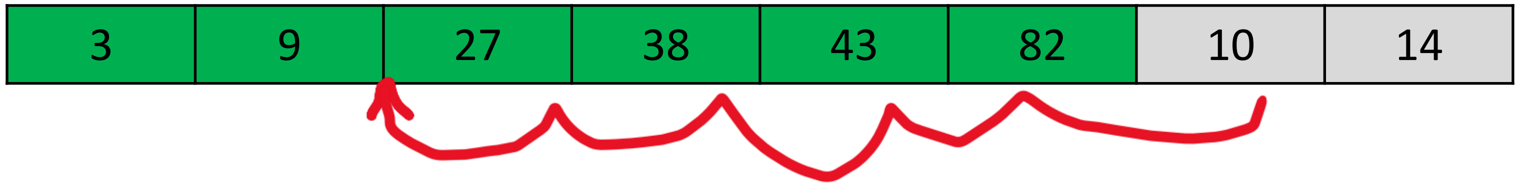
Insertion Sort

We divide our array into two sections. A sorted section, and an unsorted section. We iterate through the array, and for each iteration, we move one element from the unsorted section to the sorted section



Insertion Sort

We divide our array into two sections. A sorted section, and an unsorted section. We iterate through the array, and for each iteration, we move one element from the unsorted section to the sorted section



3	9	27	38	43	82	10	14

3	9	10	27	38	43	82	14

3	9	10	14	27	38	43	82

✓

Running time: O(n^2)

Insertion Sort

```
void insertionSort(int array[]) {
        int size = array.length;
        for (int step = 1; step < size; step++) {
                int key = array[step];
                int j = step - 1;
                // Compare key with each element on the left of it until an element smaller than
                // it is found.
                // For descending order, change key<array[j] to key>array[j].
                while (j >= 0 && key < array[j]) {
                        array[j + 1] = array[j];
                        --j;
                }
                // Place key at after the element just smaller than it.
                array[j + 1] = key;
        }
}
```

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

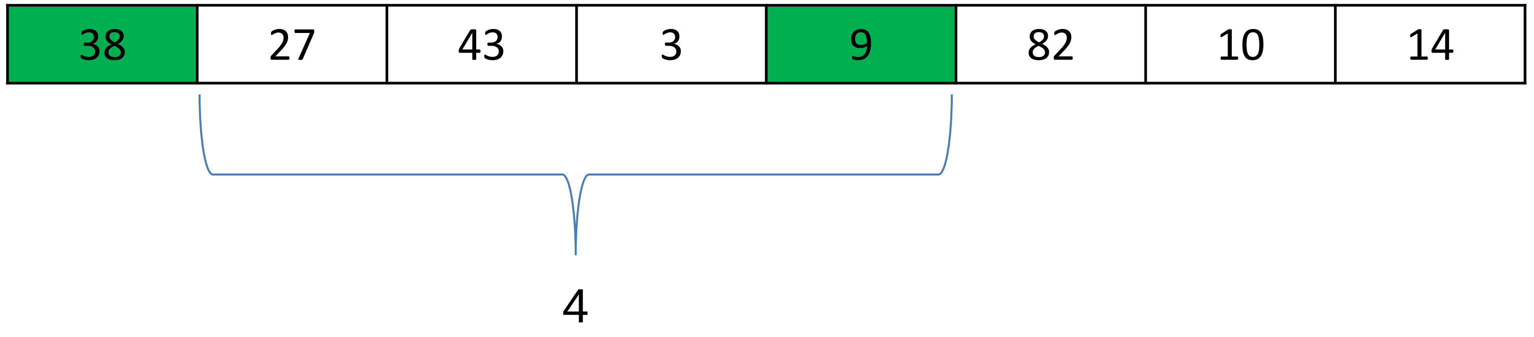
38	27	43	3	9	82	10	14

N = 8

Gap = 4

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

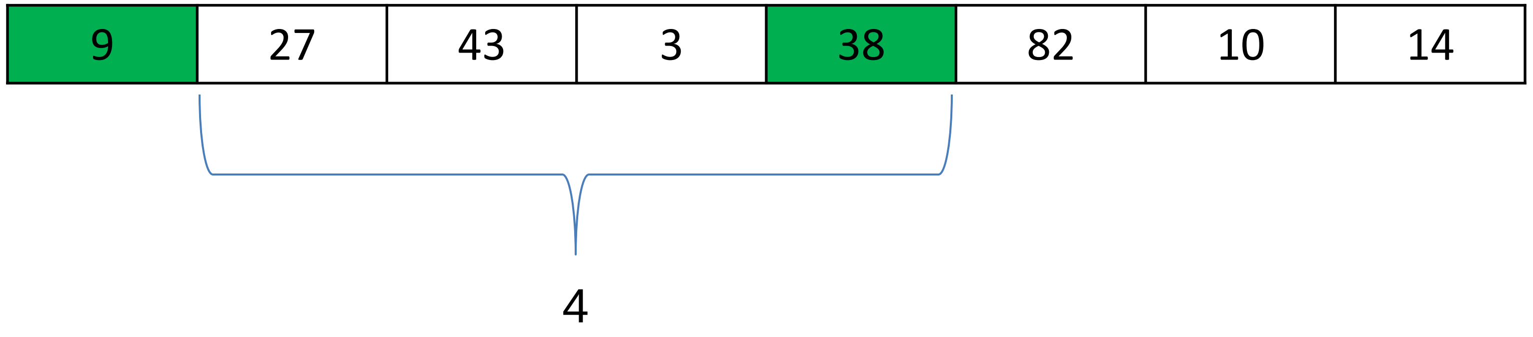


N = 8

Gap = 4

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

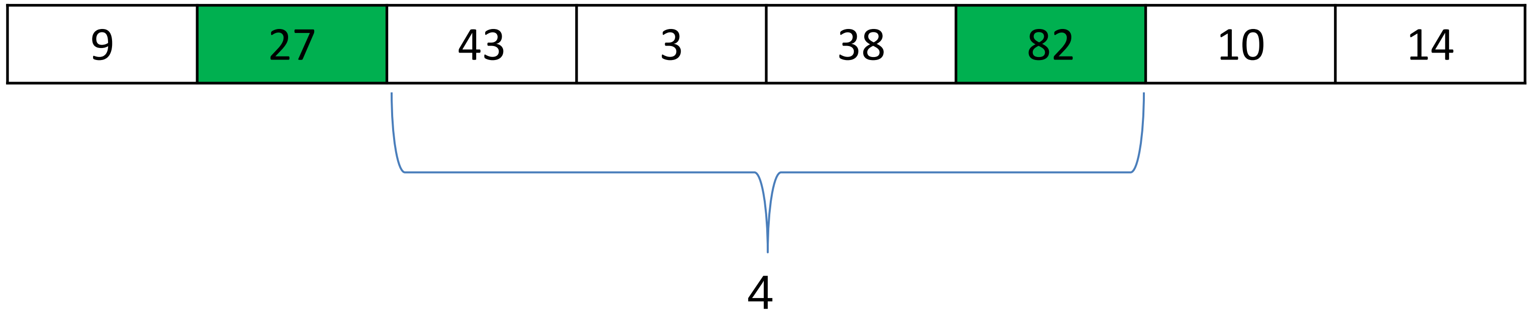


N = 8

Gap = 4

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

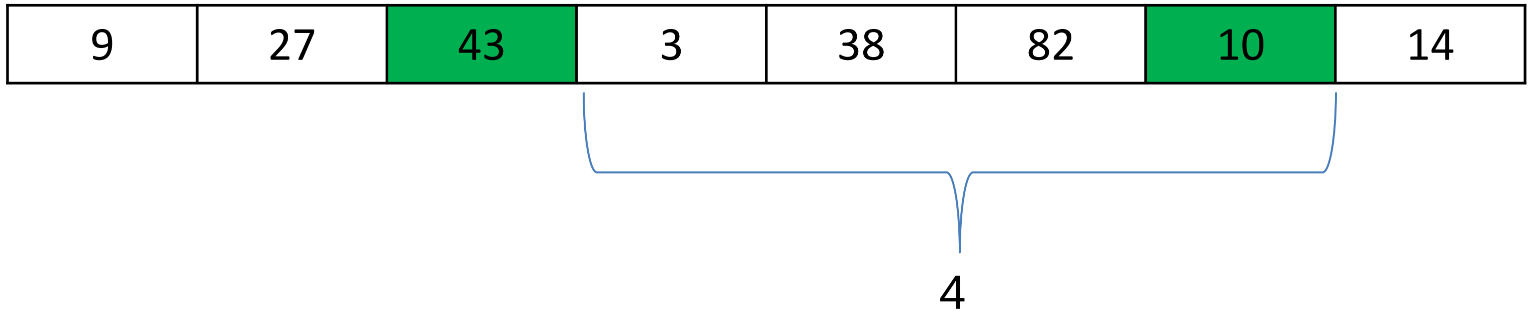


N = 8

Gap = 4

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

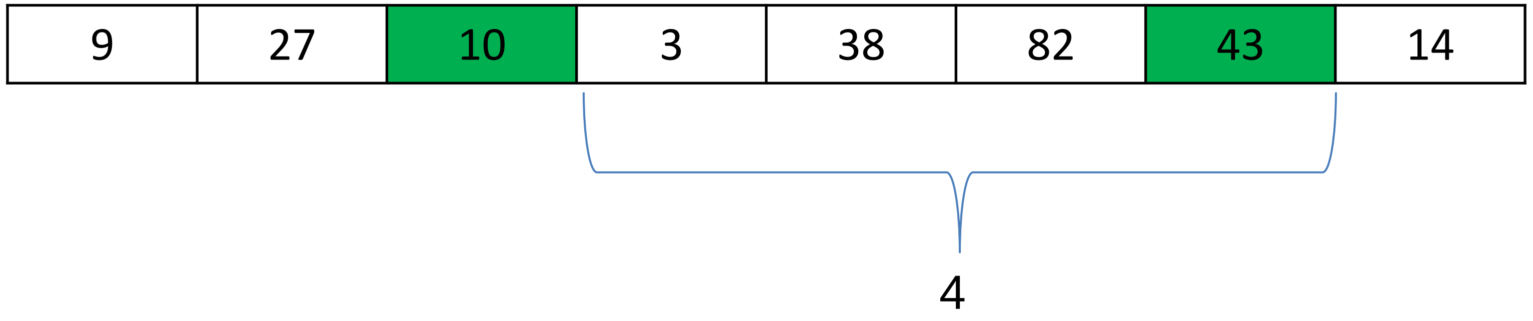


N = 8

Gap = 4

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

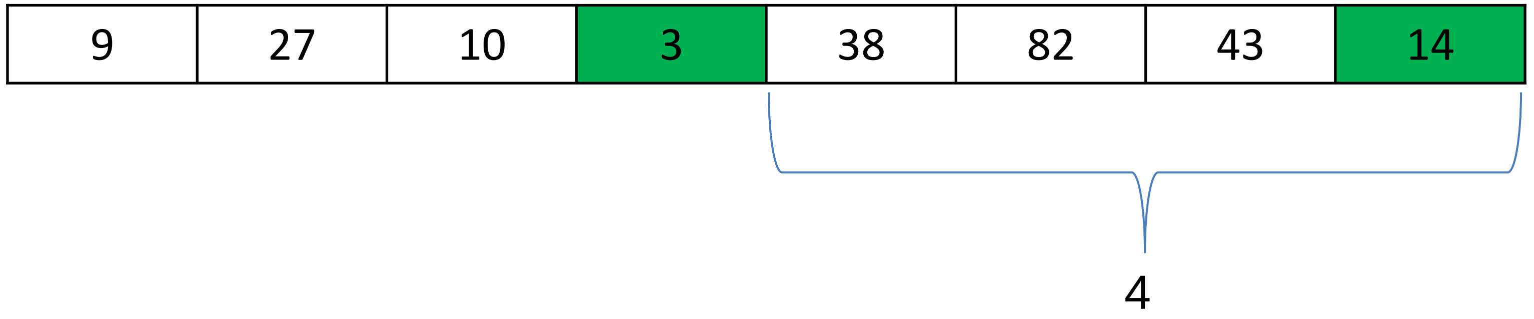


N = 8

Gap = 4

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

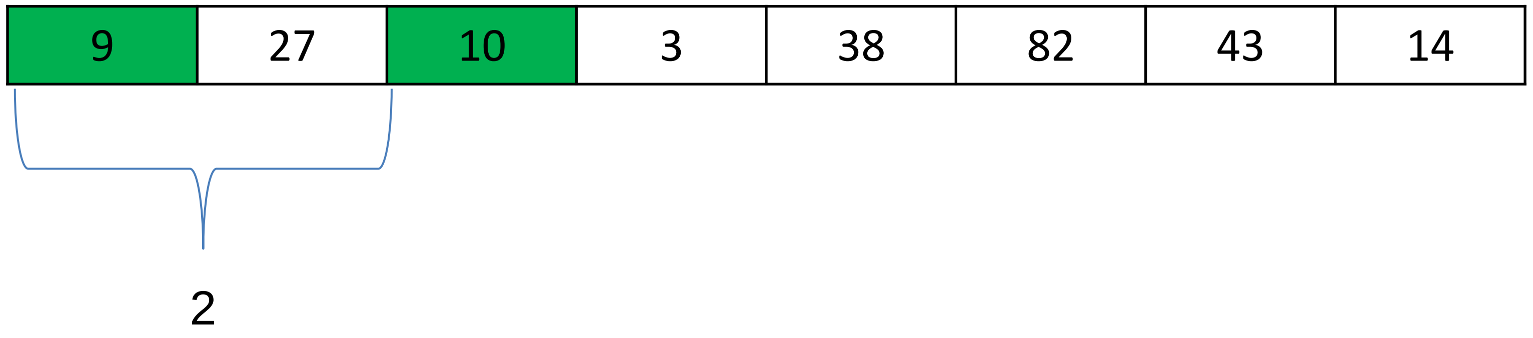


N = 8

Gap = 4

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

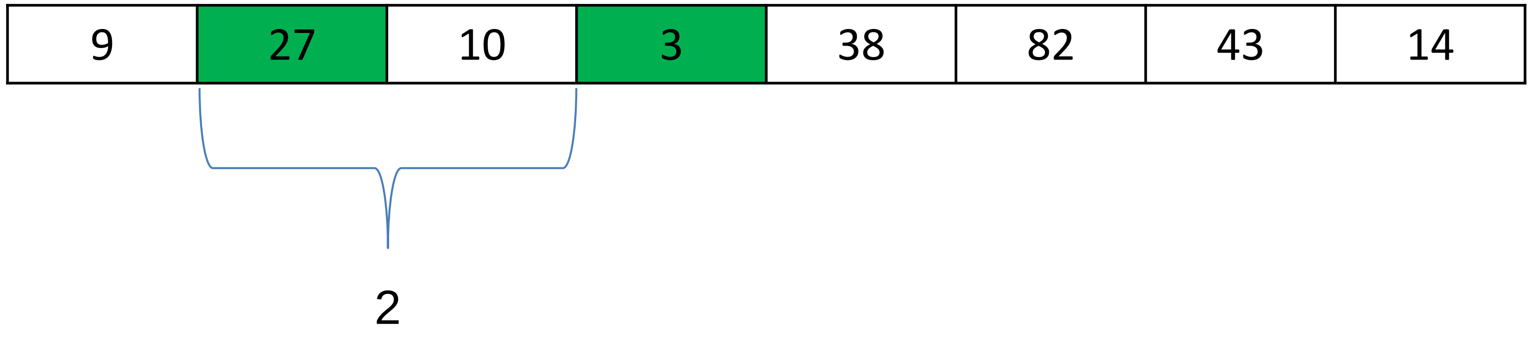


N = 8

~~Gap = 4~~
Gap = 2

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

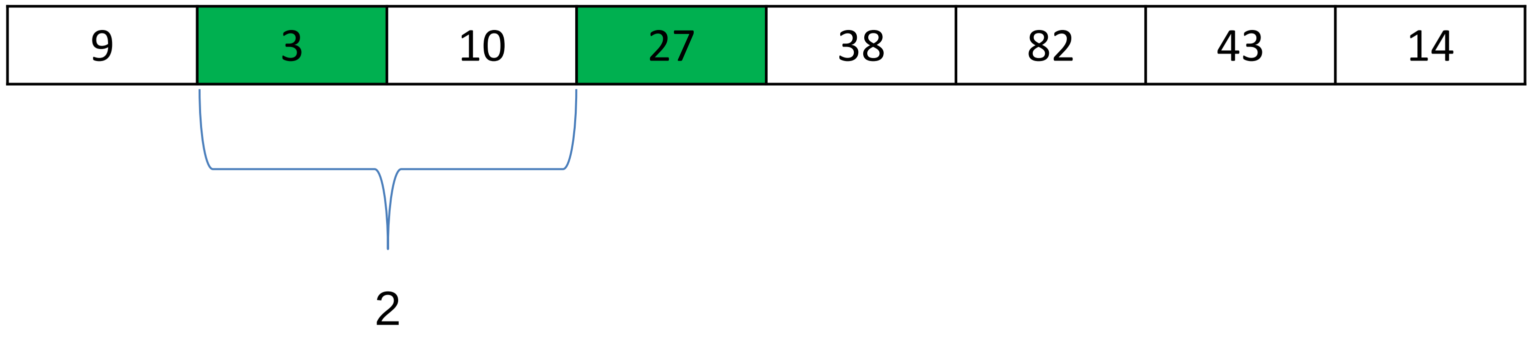


N = 8

~~Gap = 4~~
Gap = 2

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

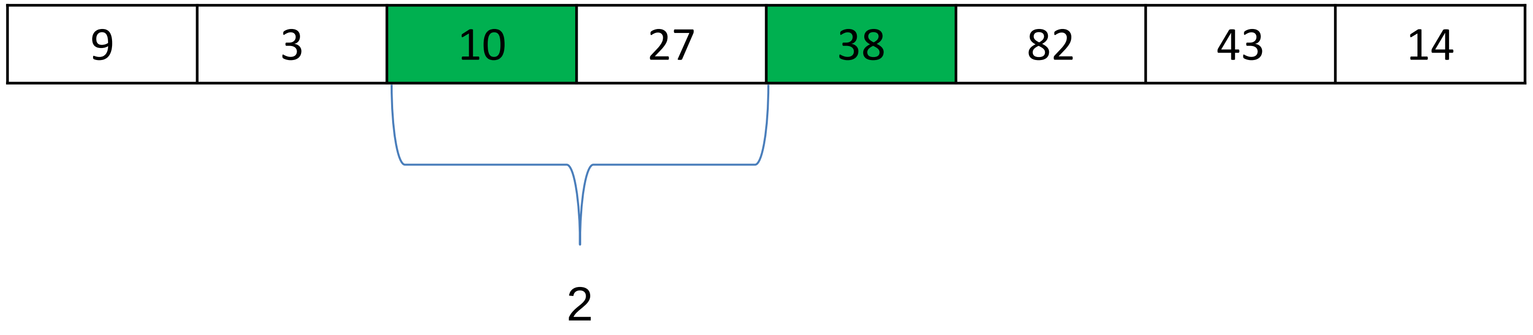


N = 8

~~Gap = 4~~
Gap = 2

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

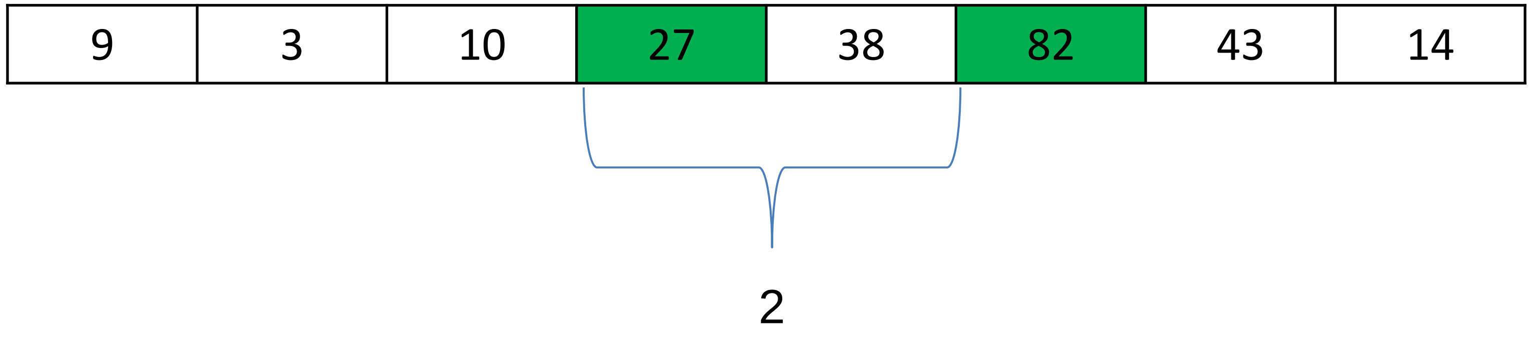


N = 8

~~Gap = 4~~
Gap = 2

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

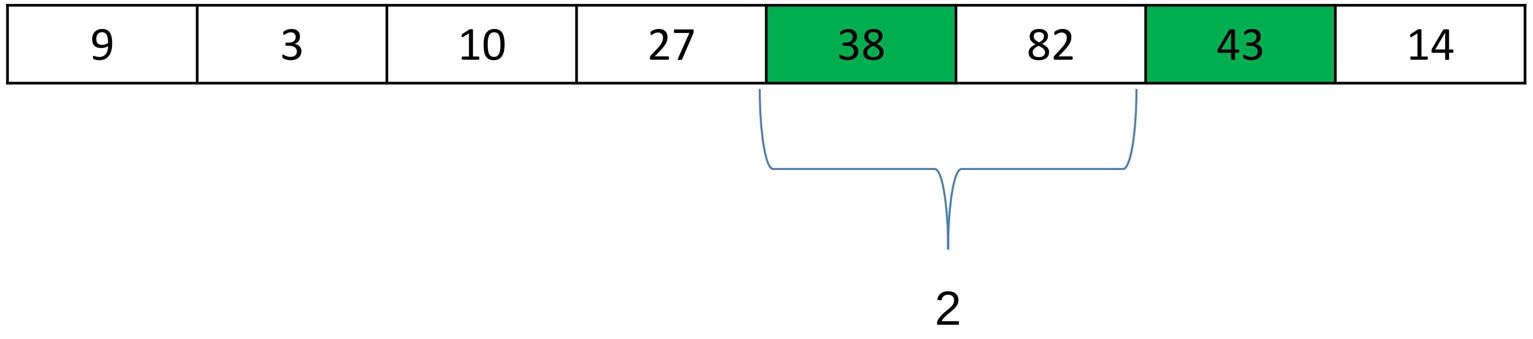


N = 8

~~Gap = 4~~
Gap = 2

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

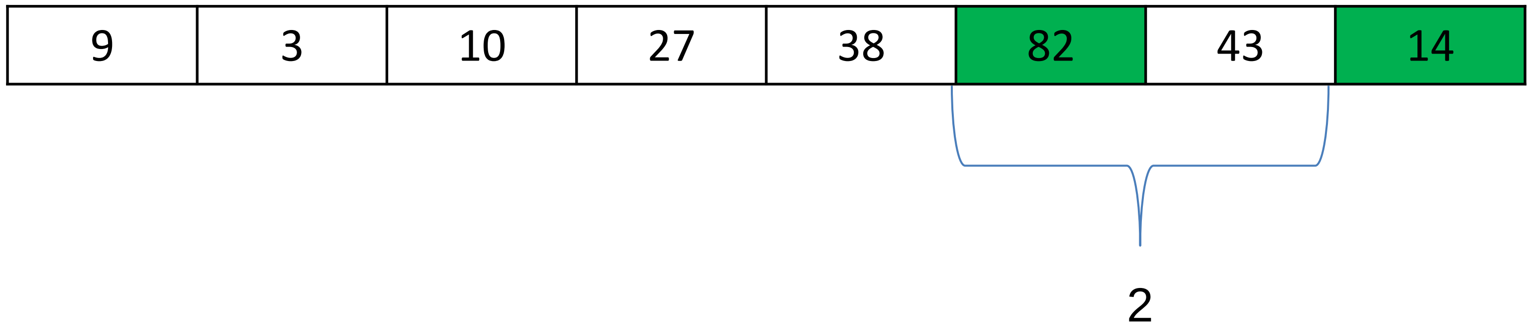


N = 8

~~Gap = 4~~
Gap = 2

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

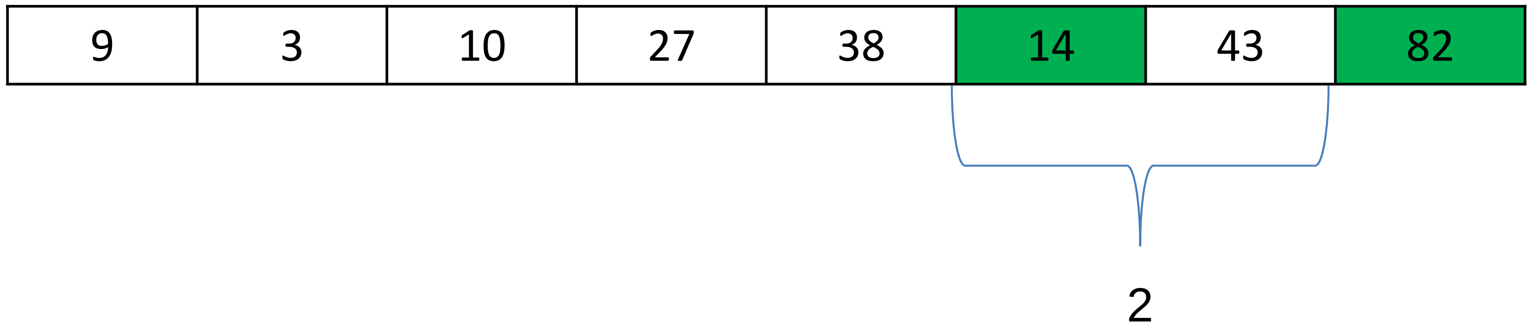


N = 8

~~Gap = 4~~
Gap = 2

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

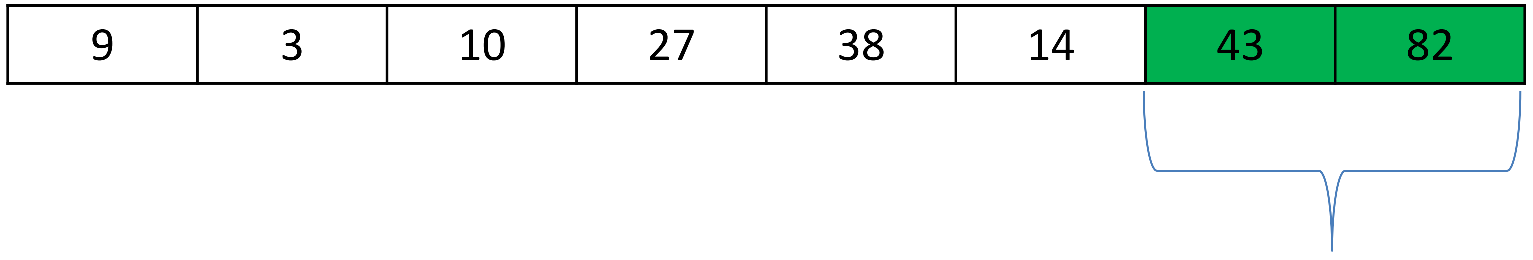


N = 8

~~Gap = 4~~
Gap = 2

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

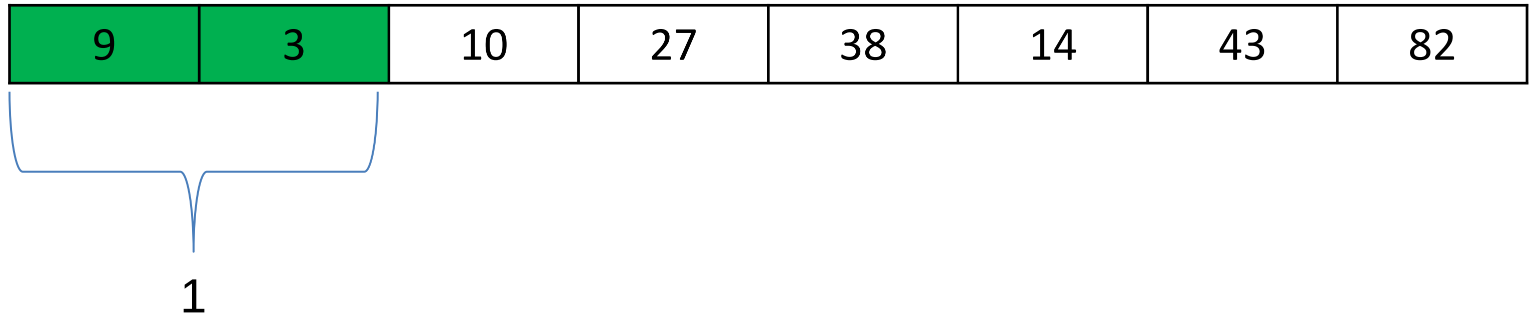


N = 8

~~Gap = 4~~
Gap = 2

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

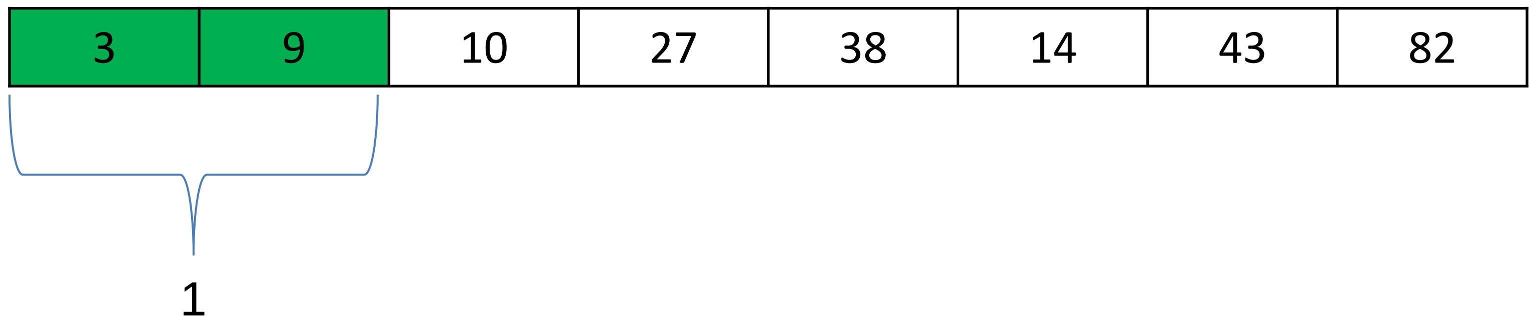


N = 8

~~Gap = 4~~
~~Gap = 2~~
Gap = 1

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

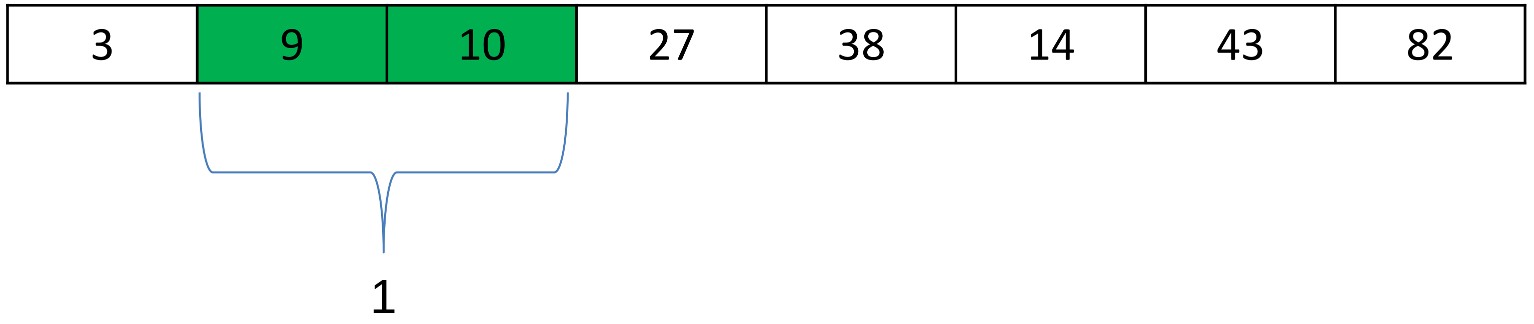


N = 8

~~Gap = 4~~
~~Gap = 2~~
Gap = 1

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

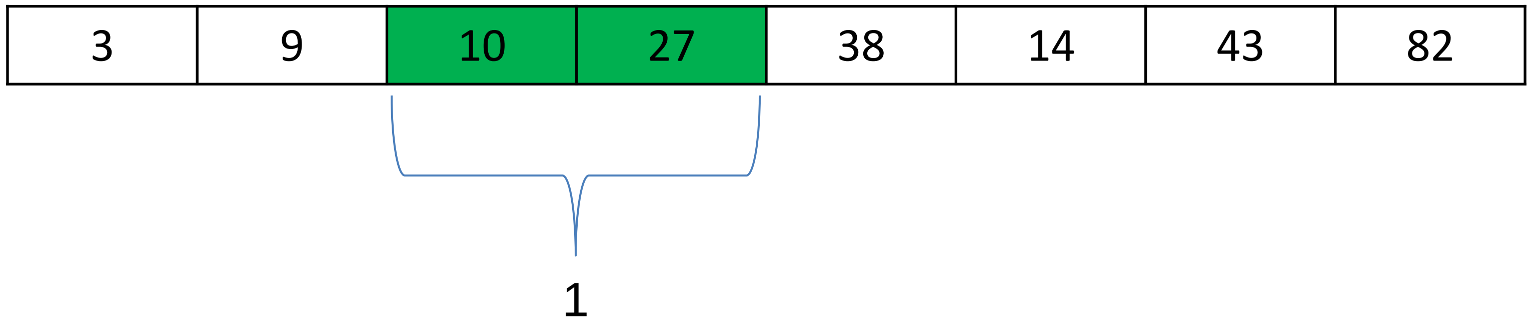


N = 8

~~Gap = 4~~
~~Gap = 2~~
Gap = 1

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

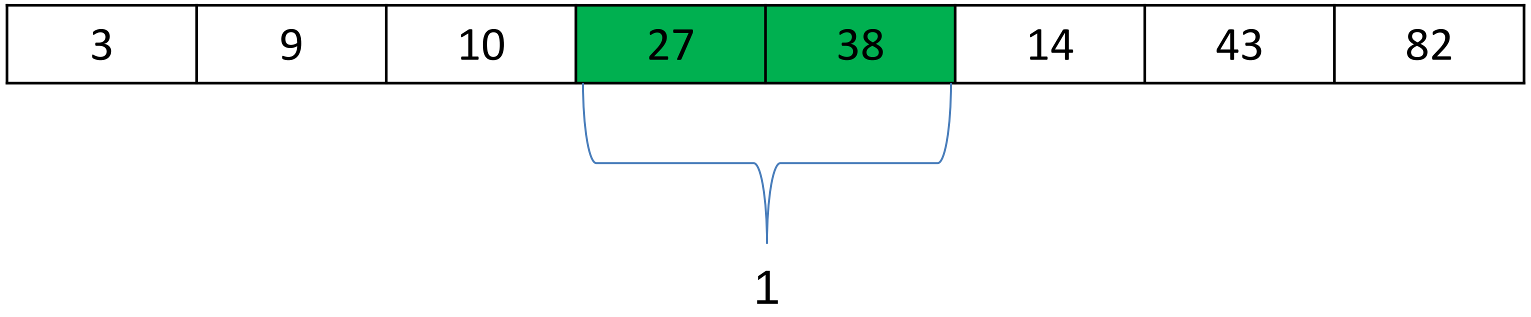


N = 8

~~Gap = 4~~
~~Gap = 2~~
Gap = 1

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

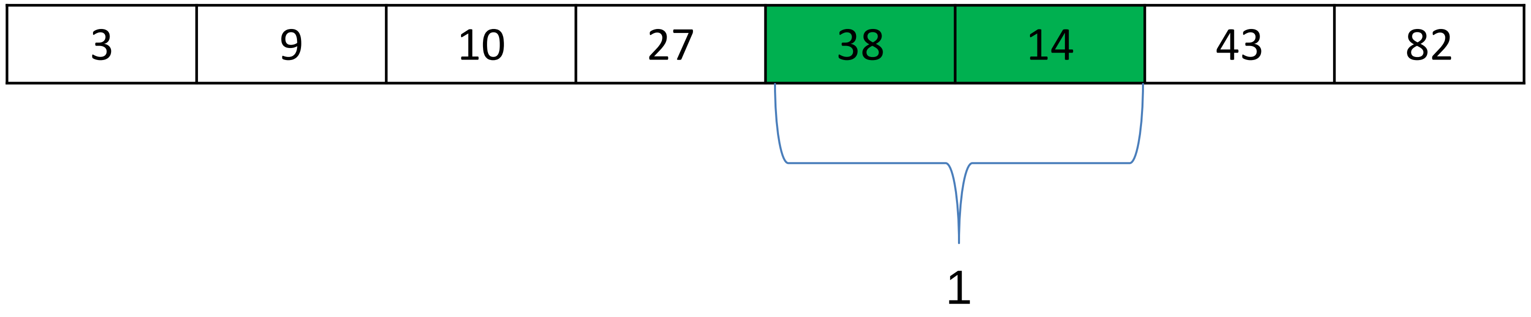


N = 8

~~Gap = 4~~
~~Gap = 2~~
Gap = 1

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

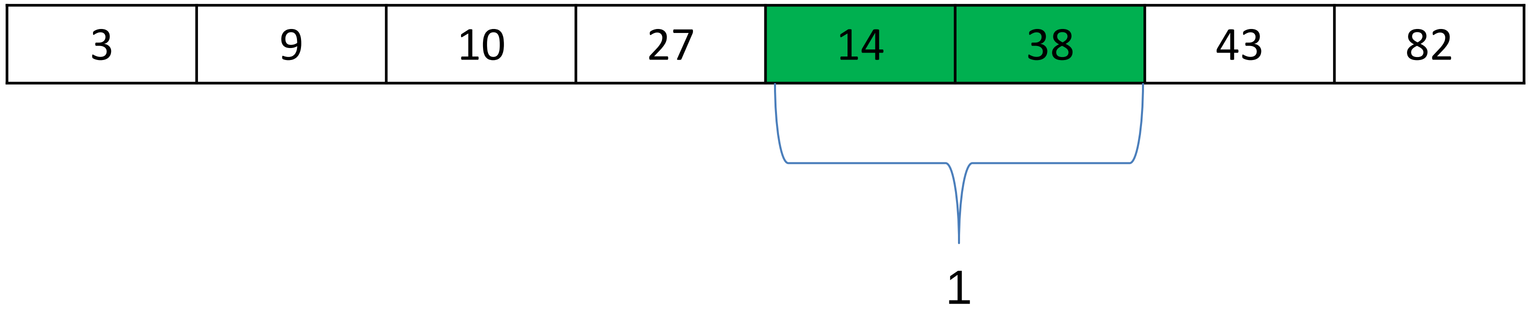


N = 8

~~Gap = 4~~
~~Gap = 2~~
Gap = 1

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

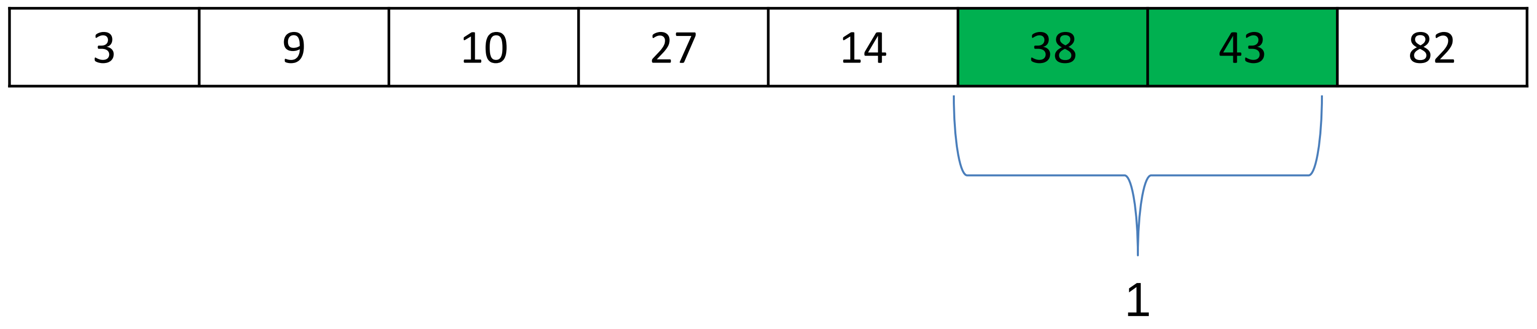


N = 8

~~Gap = 4~~
~~Gap = 2~~
Gap = 1

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

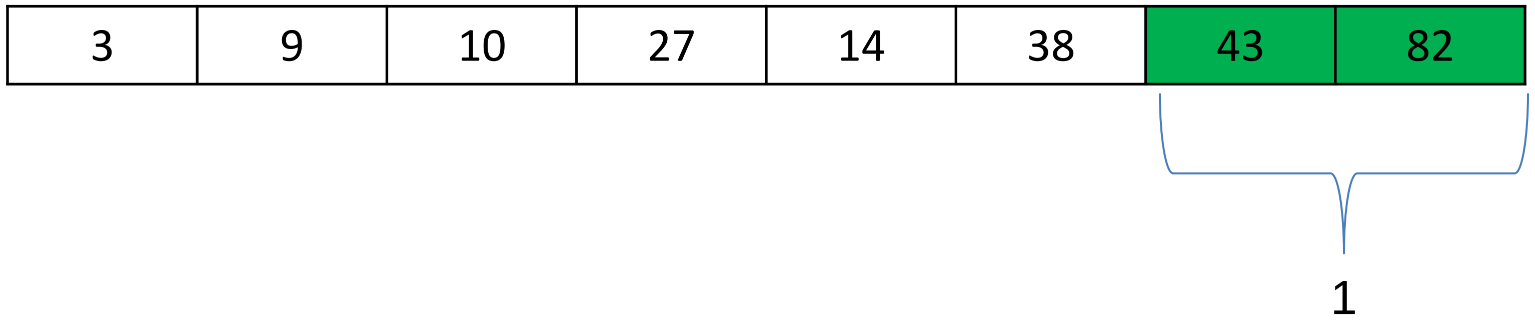


N = 8

~~Gap = 4~~
~~Gap = 2~~
Gap = 1

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.



N = 8

~~Gap = 4~~
~~Gap = 2~~
Gap = 1

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

3	9	10	27	14	38	43	82

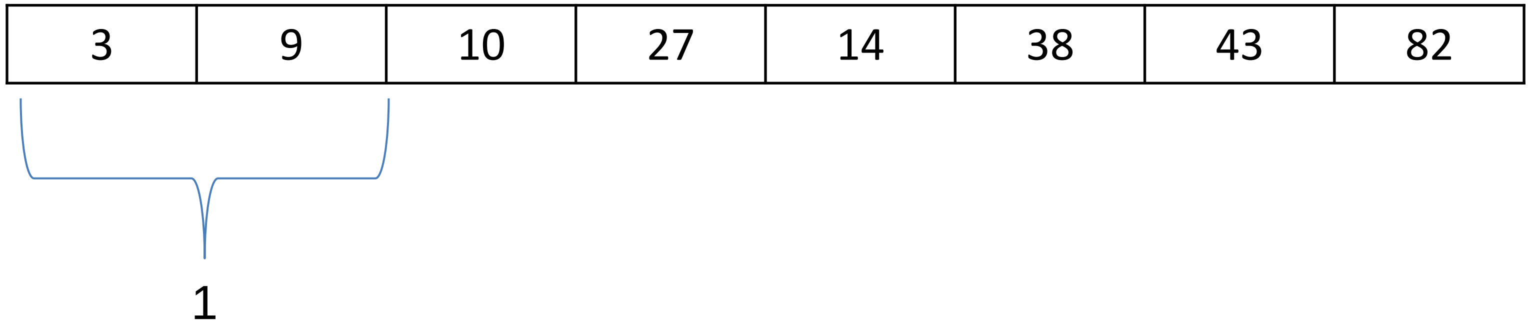
N = 8

~~Gap = 4~~
~~Gap = 2~~
Gap = 1

(do it again ??)

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

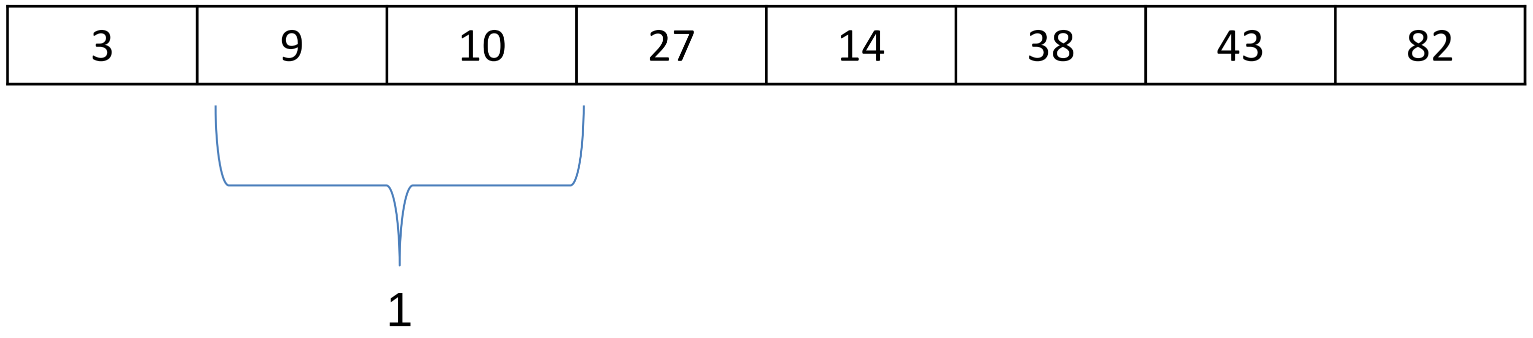


N = 8

~~Gap = 4~~
~~Gap = 2~~
Gap = 1

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.



3	9	10	27	14	38	43	82

$\underbrace{9 \quad 10}_{1}$

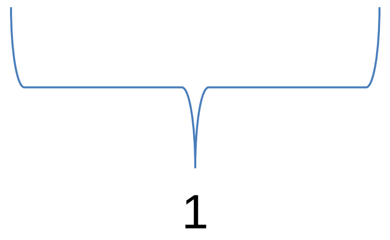
N = 8

~~Gap = 4~~
~~Gap = 2~~
Gap = 1

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

3	9	10	27	14	38	43	82

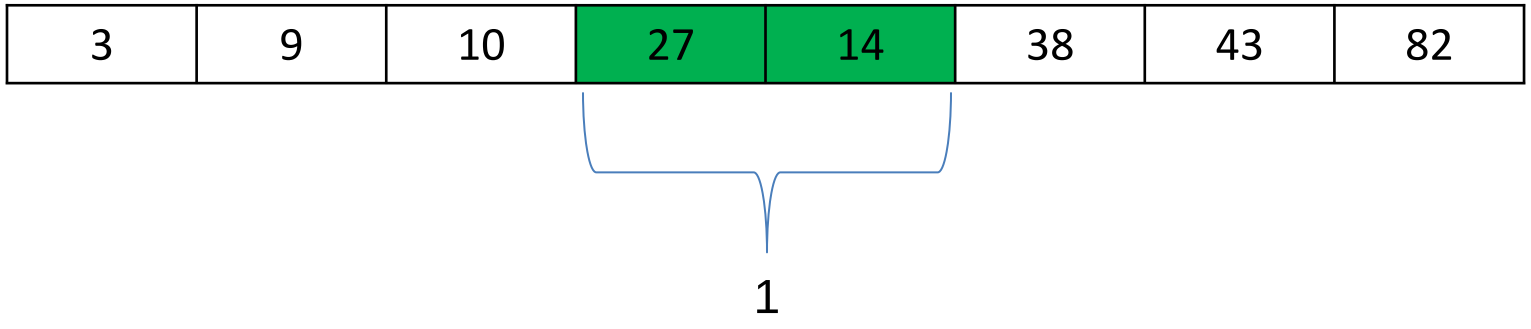


N = 8

~~Gap = 4~~
~~Gap = 2~~
Gap = 1

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

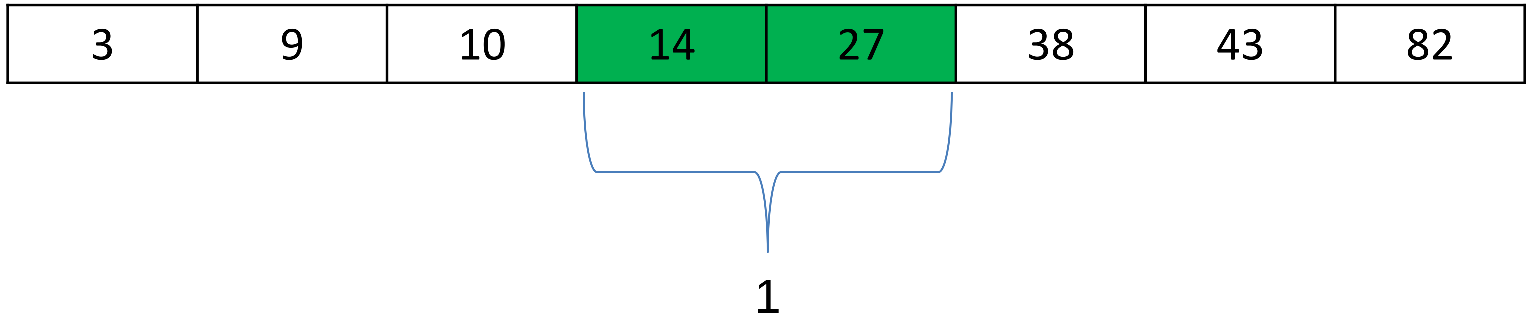


N = 8

~~Gap = 4~~
~~Gap = 2~~
Gap = 1

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.



N = 8

~~Gap = 4~~
~~Gap = 2~~
Gap = 1

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

3	9	10	14	27	38	43	82

N = 8

~~Gap = 4~~
~~Gap = 2~~
Gap = 1

✓

Running time: O(n^2)

MONTANA STATE UNIVERSITY

Cocktail Shaker Sort

Double Sided Bubble Sort

https://en.wikipedia.org/wiki/Cocktail_shaker_sort

Running time: $O(n^2)$

Does anyone have any ideas for a very bad sorting algorithm, but still works?

Does anyone have any ideas for a very bad sorting algorithm, but still works?

If we are really lucky, our algorithm is insanely fast

If we are really unlucky, our algorithm will never finish

Bogo Sort (stupid sort) randomly shuffles the array until its sorted

```
while not sorted(array):

     shuffle(array)
```

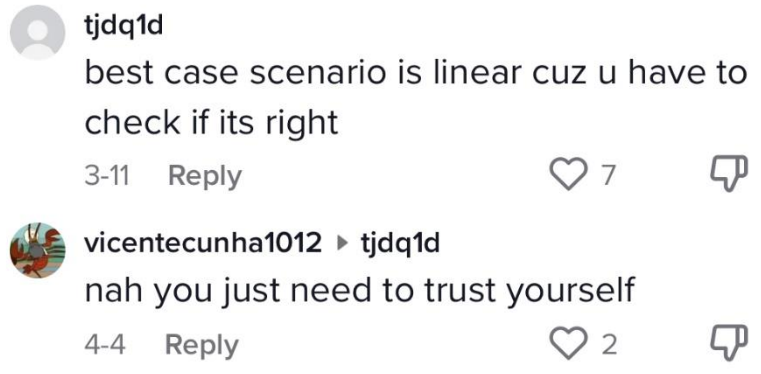
Running time: O(pain) / O(∞) if we don't keep track of permutations checked

O(n!) if we keep track of permuations

Bogo Sort (stupid sort) randomly shuffles the array until its sorted

```
while not sorted(array):

        shuffle(array)
```

Best case scenario, this is the most efficient sorting algorithm!



tjdq1d
best case scenario is linear cuz u have to check if its right
3-11 Reply 7

vicentecunha1012 ▸ tjdq1d
nah you just need to trust yourself
4-4 Reply 2

Running time: O(pain) if we don't keep track of permutations checked

O(n!) if we keep track of permutations

This sorting algorithm is a joke, please don't take this one seriously…

Sorting Algorithms Visualized

https://youtu.be/kPRA0W1kECg