AUE 8930: COMPUTING AND SIMULATION FOR AUTONOMY

Homework -3

Siddharth thorat
CUICAR

AuE 8930: Computing and Simulation for Autonomy

Instructor: Prof. Bing Li, Clemson University, Department of Automotive Engineering

Part-A

Question 1 (10')

Given an array of integers, find two numbers in it such that they can add up to a specific number. You may assume there are exactly one solution, you can't use the same element twice. (Only time-complexity optimized solution gets full grade)
Example:
Given [2, 7, 11, 4], Target = 13.
The answer is 2 and 11.

Modify the "solution" function in the question1.py.
(Analyze your time complexity)

```
In [3]: runfile('C:/Users/siddh/OneDrive/Pictures/Screenshots/
Documents/Course/AuE 8930/HW3/q!.py', wdir='C:/Users/siddh/OneDr
Pictures/Screenshots/Documents/Course/AuE 8930/HW3')
13

In [4]:
```

Question 2 (10')

Given a binary tree, find the max depth of it. Modify the "solution" function in the question2.py (Analyze your time complexity, and only time-complexity optimized solution gets full grade)

```
In [5]: runfile('C:/Users/siddh/OneDrive/Pictures/Screenshots/
Documents/Course/AuE 8930/HW3/q2.py', wdir='C:/Users/siddh/OneDrive/
Pictures/Screenshots/Documents/Course/AuE 8930/HW3')
3
```

Question 3 (5')

You are given two non-empty linked lists representing two non-negative integers. The digits are stored in reverse order and each of their nodes contain a single digit. Add the two numbers and return it as a linked list.

You may assume the two numbers do not contain any leading zero, except the number 0 itself.

```
In [8]: runfile('C:/Users/siddh/OneDrive/Desktop/HPC 3/question3.py'
wdir='C:/Users/siddh/OneDrive/Desktop/HPC 3')
First List is
3
4
2

Second List is
4
6
5

Resultant list is
7
0
8

In [9]:
```

Question 4 (5’)

Given a string s, find the length of the longest substring without repeating characters. You can expect the string length is less than 100, and only contains English letters.
Example 1:
Input: s = "abcabcbb"
Output: 3
Explanation: The answer is "abc", with the length of 3.

Modify the “solution” class in question4.py, you may design your input to test it.

```
In [13]: runfile('C:/Users/siddh/OneDrive/Desktop/HPC 3/question4.py',
wdir='C:/Users/siddh/OneDrive/Desktop/HPC 3')

Enter String :siddharth
5
```

```
In [14]: runfile('C:/Users/siddh/OneDrive/Desktop/HPC 3/question4.py',
wdir='C:/Users/siddh/OneDrive/Desktop/HPC 3')

Enter String :asdfghjk
8
```

Question 5 (5’)

Given a string s containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.
An input string is valid if:

- Open brackets must be closed by the same type of brackets.
- Open brackets must be closed in the correct order.
- Every close bracket has a corresponding open bracket of the same type.

Modify the “solution” function in the question5.py. (Analyze your time complexity)

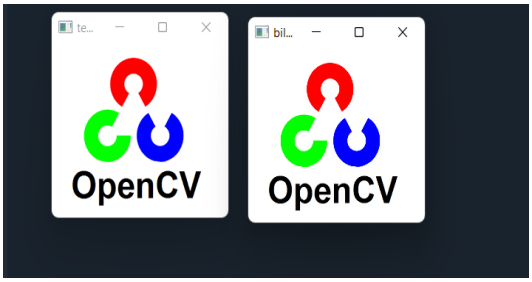
```
if __name__ == '__main__':
    print(solution('([]){'))
    print(solution('[]'))
```

```
In [20]: runfile('C:/Users/siddh/OneDrive/Desktop/HPC 3/untitled1.py
wdir='C:/Users/siddh/OneDrive/Desktop/HPC 3')
False
True

In [21]:
```

Question 6 (5’)

Use OpenCV to do a bilateral filter to an image, modify from question6.py, you may use your favorite image, visualize the images before and after the filtering using matplotlib.

```
IPython 8.1.1 -- An enhanced Interactive Python.

In [1]: runfile('C:/Users/siddh/OneDrive/Desktop/HPC 3/question6.py',
wdir='C:/Users/siddh/OneDrive/Desktop/HPC 3')
[[[255 255 255]
  [255 255 255]
  [255 255 255]
  ...
  [255 255 255]
  [255 255 255]
  [255 255 255]]

 [[255 255 255]
  [255 255 255]
  [255 255 255]
  ...
  [255 255 255]
  [255 255 255]
  [255 255 255]]
```



Question 7 (10’)

Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum. (Note: A leaf is a node with no children.)

```
In [9]: runfile('C:/Users/siddh/OneDrive/Desktop/HPC 3/Question_7.py',
wdir='C:/Users/siddh/OneDrive/Desktop/HPC 3')
There is no root-to-leaf path with given sum
```

Question 8 (10’)

Given two strings s and t, return true *if* t *is an anagram of* s*, and* false *otherwise*.

An **Anagram** is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

```
    s = "anagram"
    t = "nagaram"
    if(Solution().isAnagram(s, t)):
      print("The two strings are anagram of each other [true]")
    else:
      print("The two strings are not anagram of each other [False]")
```

```
In [12]: runfile('C:/Users/siddh/OneDrive/Desktop/HPC 3/question8.py',
wdir='C:/Users/siddh/OneDrive/Desktop/HPC 3')
The two strings are anagram of each other [true]

In [13]:
```

```
if __name__ == '__main__':
    s = "anagram"
    t = "cat"
    if(Solution().isAnagram(s, t)):
      print("The two strings are anagram of each other [true]")
    else:
      print("The two strings are not anagram of each other [False]")
```

```
In [13]: runfile('C:/Users/siddh/OneDrive/Desktop/HPC 3/question8.py',
wdir='C:/Users/siddh/OneDrive/Desktop/HPC 3')
The two strings are not anagram of each other [False]
```

Part-B

Demo existing and revise search algorithms (40')

- Reference code repo:
  https://github.com/fengziyue/CU-Computing-Autonomy/tree/master/Homework3/map-path-search

[a] For this question, you have the existing reference:

Occupancy gridmap class library:

Homework3/map-path-search/gridmap.py

Occupancy gridmap-based A* (A-start path searching algorithm) implementation:

Homework3/map-path-search/a_star_occupancy.py

As the default, you can use the 8 connectivity for this whole question.

1) Demo existing reference and get to know the behaviour of its path search. (2')

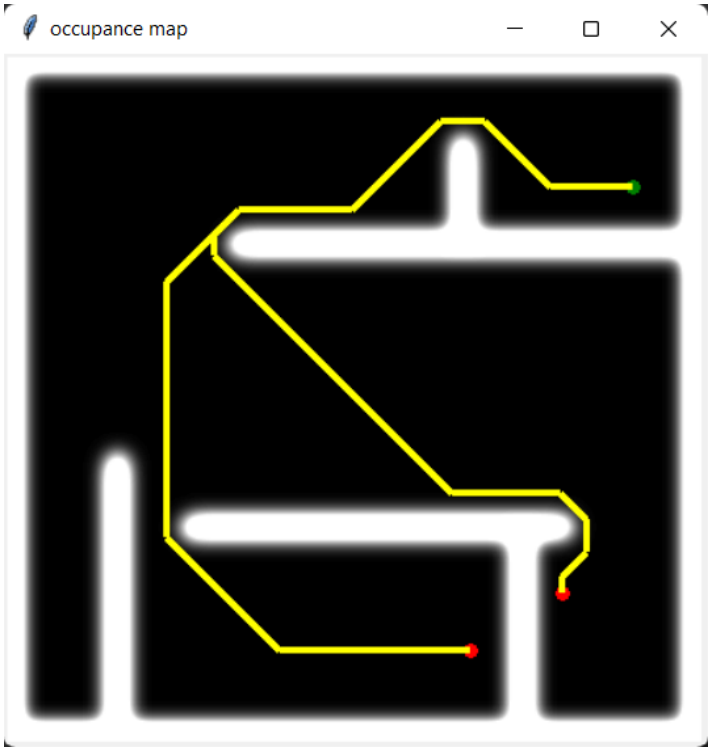
The demo run file is: *examples/occupancy_map_8n.py*

2) Implement occupancy gridmap-based Dijkstra for same functionality as (1a) (18')

If you prefer, you can use this as the template to revise:

Homework3/map-path-search/a_star_occupancy.py

1.



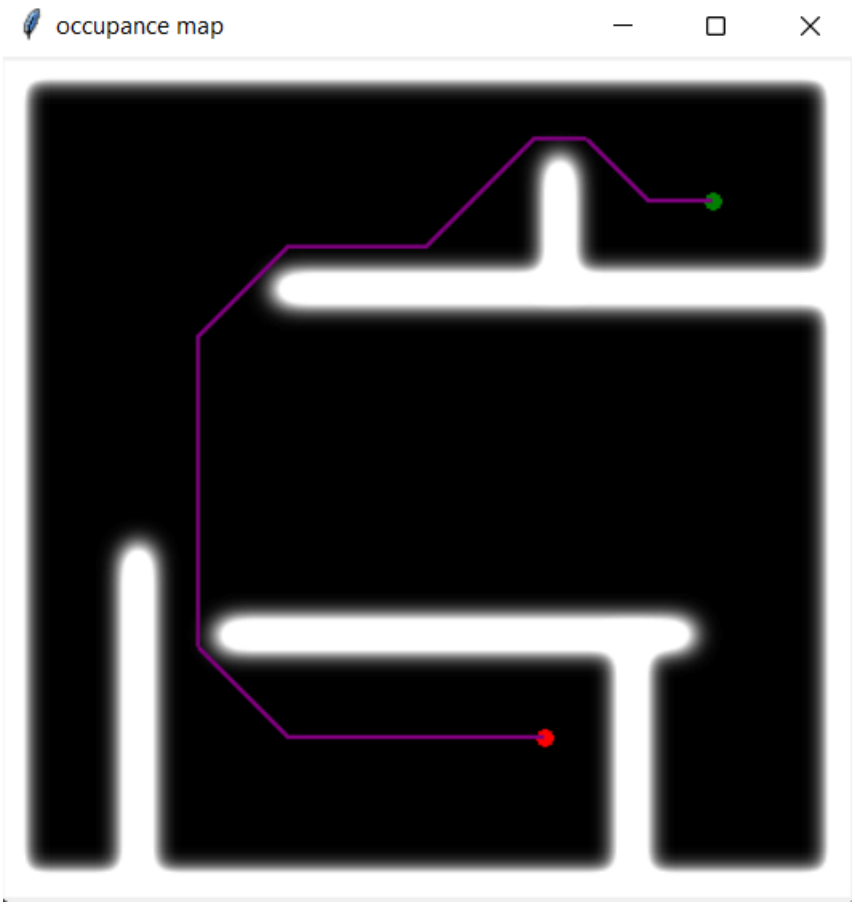
2.

```
def dijkstra_occupancy(start_m, goal_m, gmap, movement='8N', occupancy_cost_factor=3):
    """
    dijkstra for 2D occupancy grid.

    :param start_m: start node (x, y) in meters
    :param goal_m: goal node (x, y) in meters
    :param gmap: the grid map
    :param movement: select between 4-connectivity ('4N') and 8-connectivity ('8N', default)
    :param occupancy_cost_factor: a number the will be multiplied by the occupancy probability
        of a grid map cell to give the additional movement cost to this cell (default: 3).

    :return: a tuple that contains: (the resulting path in meters, the resulting path in data array indices)
    """

    # get array indices of start and goal
    start = gmap.get_index_from_coordinates(start_m[0], start_m[1])
    goal = gmap.get_index_from_coordinates(goal_m[0], goal_m[1])
```

```
    # check if start and goal nodes correspond to free spaces
    if gmap.is_occupied_idx(start):
        raise Exception('Start node is not traversable')

    if gmap.is_occupied_idx(goal):
        raise Exception('Goal node is not traversable')

    # add start node to front
    # front is a list of (total estimated cost to goal, total cost from start to node, node, previous node)
    start_node_cost = 0
    start_node_estimated_cost_to_goal = dist2d(start, goal) + start_node_cost
    front = [(start_node_estimated_cost_to_goal, start_node_cost, start, None)]

    # use a dictionary to remember where we came from in order to reconstruct the path later on
    came_from = {}

    # get possible movements
    if movement == '4N':
        movements = _get_movements_4n()
    elif movement == '8N':
        movements = _get_movements_8n()
    else:
        raise ValueError('Unknown movement')

    # while there are elements to investigate in our front.
    while front:
        # get smallest item and remove from front.
        element = heappop(front)

        # if this has been visited already, skip it
        total_cost, cost, pos, previous = element
        if gmap.is_visited_idx(pos):
```

```
            continue

        # now it has been visited, mark with cost
        gmap.mark_visited_idx(pos)

        # set its previous node
        came_from[pos] = previous

        # if the goal has been reached, we are done!
        if pos == goal:
            break

        # check all neighbors
        for dx, dy, deltacost in movements:
            # determine new position
            new_x = pos[0] + dx
            new_y = pos[1] + dy
            new_pos = (new_x, new_y)

            # check whether new position is inside the map
            # if not, skip node
            if not gmap.is_inside_idx(new_pos):
                continue

            # add node to front if it was not visited before and is not an obstacle
            if (not gmap.is_visited_idx(new_pos)) and (not gmap.is_occupied_idx(new_pos)):
                potential_function_cost = gmap.get_data_idx(new_pos)*occupancy_cost_factor
                new_cost = cost + deltacost + potential_function_cost
                new_total_cost_to_goal = new_cost + potential_function_cost

                heappush(front, (new_total_cost_to_goal, new_cost, new_pos, pos))

    # reconstruct path backwards (only if we reached the goal)
    path = []
```

```
    path_idx = []
    if pos == goal:
        while pos:
            path_idx.append(pos)
            # transform array indices to meters
            pos_m_x, pos_m_y = gmap.get_coordinates_from_index(pos[0], pos[1])
            path.append((pos_m_x, pos_m_y))
            pos = came_from[pos]

        # reverse so that path is from start to goal.
        path.reverse()
        path_idx.reverse()

    return path, path_idx
```



[b] For this question, you have the existing reference:

Quadtree-map class library:

Homework3/map-path-search/quadtreemap.py

Quadtree map-based Dijkstra path searching algorithm implementation:

Homework3/map-path-search/dijkstra_quadtree.py

3) Demo existing reference and get to know the behaviour of its path search. (2')

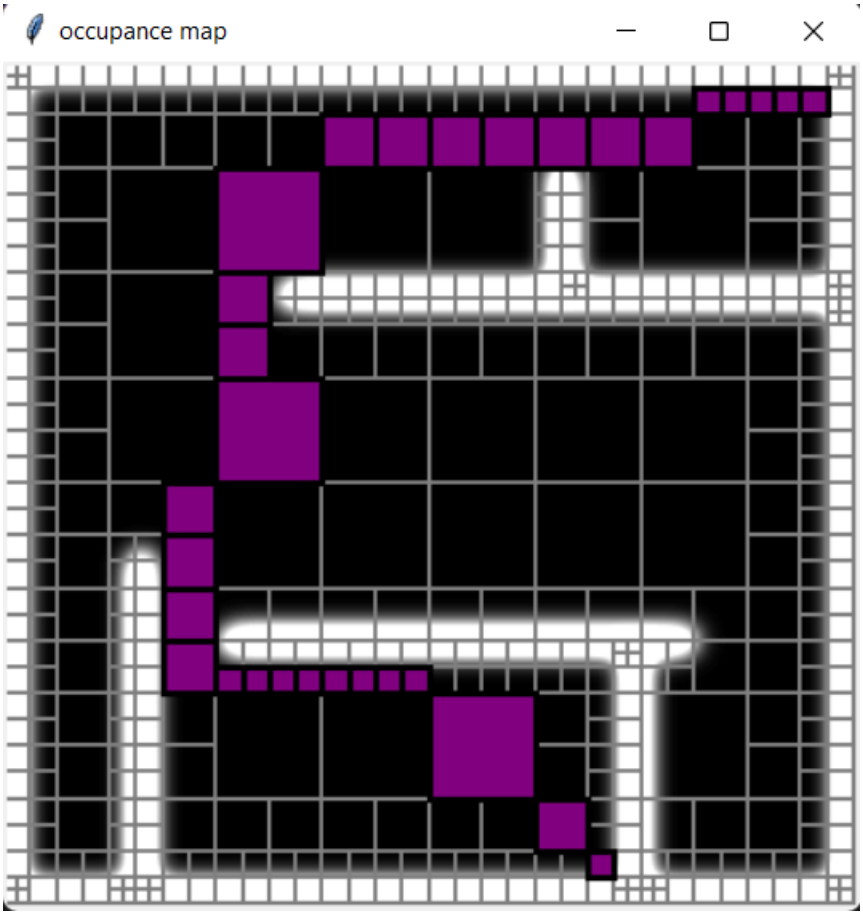
The demo run file is: *examples/quadtree_map_8n.py*

4) Implement Quadtree map-based A* for same functionality as (2a) (18')

If you prefer, you can use this as the template to revise:

Homework3/map-path-search/dijkstra_quadtree.py

3)

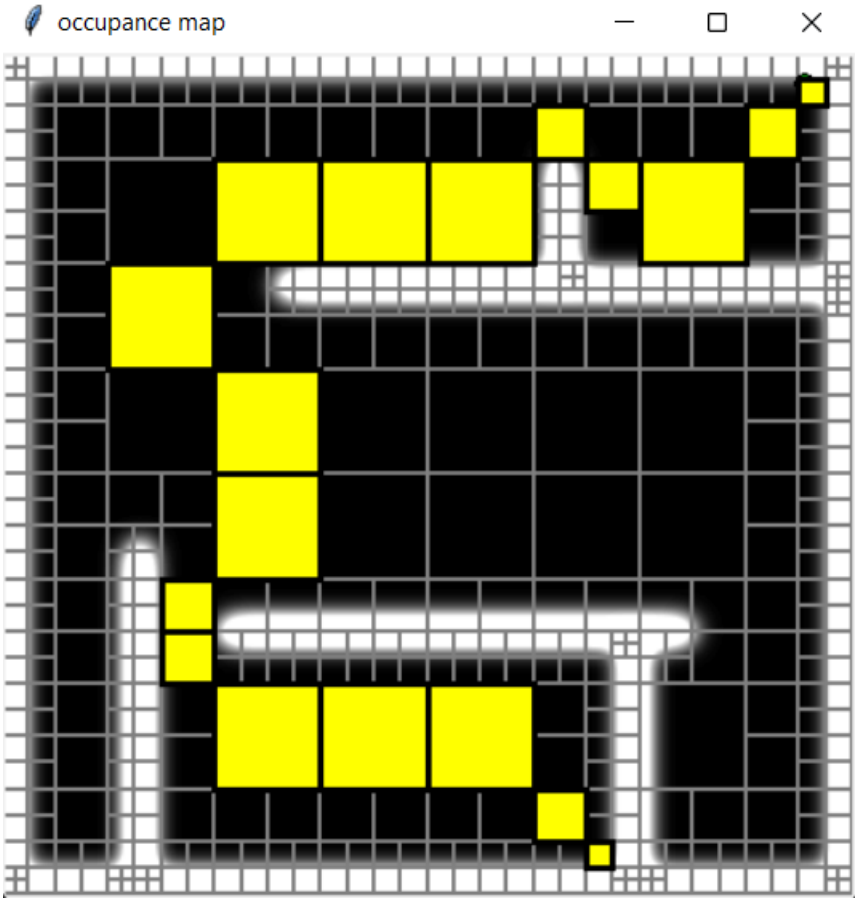


4)

```
def a_star_quadtree(start_m, goal_m, qtm, movement='8n', occupancy_cost_factor=3):
    path_record = {}
    candidates = queue.PriorityQueue()

    # get array indices of start and goal
    start = qtm.quadtree.searchTileByIdx(quadtreemap.Point(start_m[0], start_m[1]))
```

```
    goal = qtm.quadtree.searchTileByIdx(quadtreemap.Point(goal_m[0], goal_m[1]))

    # check if start and goal nodes correspond to free spaces
    if not start or start.tile_points:
        raise Exception('Start node is not traversable')
    if not goal or goal.tile_points:
        raise Exception('Goal node is not traversable')

    candidates.put((0, None, start))   # store (distance, previous-tile, current-tile)
    while candidates:
        dis, prev_node, curr_node = candidates.get()
        # print(curr_node, "\t", goal)
        if curr_node == goal:
            # print(True)
            path_record[curr_node] = prev_node
            break
        if curr_node in path_record:
            continue
        path_record[curr_node] = prev_node

        # get possible movements
        if movement == '4N':
            movements = _get_movements_4n(qtm, curr_node)
        elif movement == '8N':
            movements = _get_movements_8n(qtm, curr_node)
        else:
            raise ValueError('Unknown movement')

        # check all neighbors
        for til, deltacost in movements:
            # check whether new position is inside the map or is an obstacle
            # if not, skip node
            if til.tile_points:
                continue
```

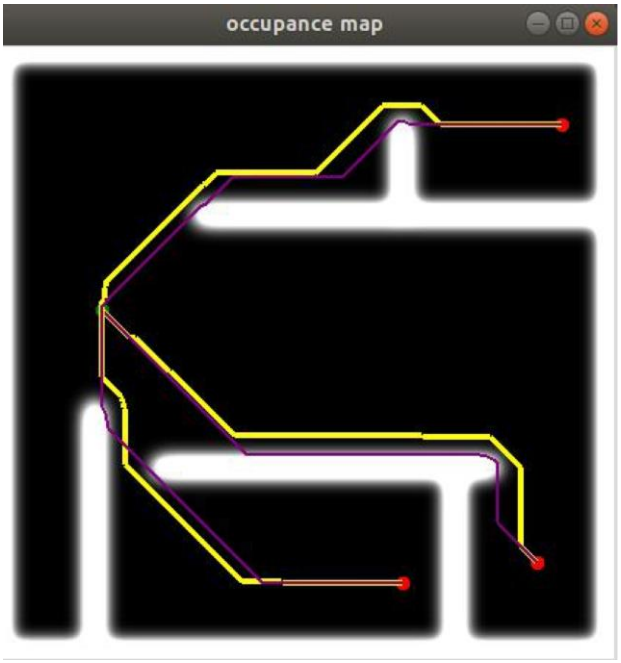
```
            if til not in path_record:
                candidates.put((dis +
deltacost+quadtreemap.Point.disOf2Points(til.getCenter(), goal.getCenter()), curr_node,
til))
    # reconstruct path backwards (only if we reached the goal)
    path = []
    path_idx = []
    # print(len(path_record))
    # print(path_record)
    if goal in path_record:
        node = goal
        while node:
            path_idx.append(node)
            # transform array indices to meters
            # node_m_x, node_m_y = gmap.get_coordinates_from_index(node[0], node[1])
            # path.append((node_m_x, node_m_y))
            node = path_record[node]
        # reverse so that path is from start to goal.
        path.reverse()
        path_idx.reverse()

    # print("path_idx len: ", len(path_idx))
    # print("path_idx:\n", path_idx)
    return path, path_idx
```

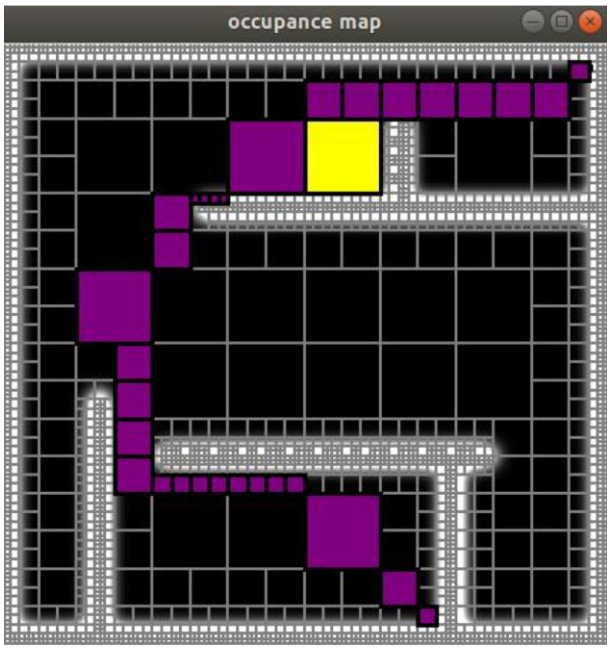


[c] Extra credits (optional to complete):

- 5) Try a few of different granularity, and describe the potential affect (2');
- 6) Show both Dijkstra and A* algorithms into the GUI for the same mouse events (2');
- 7) Analyze your time complexity of each algorithm (2');

Below are some visualization hints for your references:





6)