GAME BASED LEARNING OF SORTING ALGORITHMS

Software Engineering Lab (CS243)

January – April (2017)

BLACK BOX TESTING DOCUMENT

Instructor

Dr. Samit Bhattacharya

CSE Department, IITG

Group Number: 02

Project Team Members

150101003: Abhishek Kumar

150101045: Patoliya Meetkumar Krushnadas

150101054: Saket Sanjay Agrawal

Index

1. SELECTION SORT:

Function Description:

The algorithm proceeds by **finding the smallest** (or largest, depending on sorting order) element in the unsorted sub-list, exchanging (**swapping**) it with the leftmost unsorted element (putting it in sorted order), and moving the sub-list boundaries one element to the right.

Orange color box for leftmost unsorted element.

Green color boxes are for sorted element.

Consider an array with five entries,

35	20	19	16	45
1	2	3	4	5

Here, smallest element from the sub-list [35, 20, 19, 16, 45] will be selected and swapped with entry at index number 1.So, smallest element is 16 at index number 4 ,after swapping (after first **valid** move),

Temp = List [1]

List [1] = List [4]

List [4] = Temp

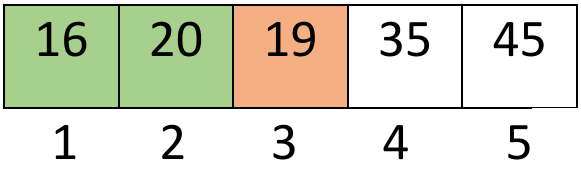
16	20	19	35	45
1	2	3	4	5

After second **valid** move,

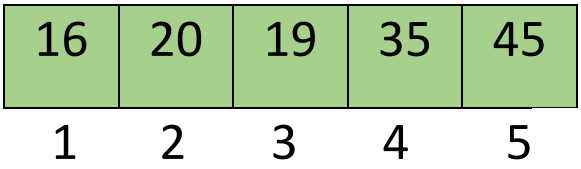
Temp = List [2]

List [2] = List [3]

List [3] = Temp



No more moves are possible and hence array is sorted,



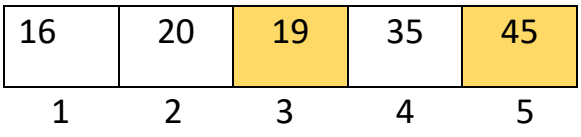
"YOU WON" will be displayed on the screen.

Equivalence class partitioning and Test cases:

1. **System will detect number of touches by user at a particular time instance. Therefore, equivalence class partitioning based on, "how many boxes are touched by user".**
   (Here, yellow color boxes denote that the user touches them).

   ## Equivalence classes:

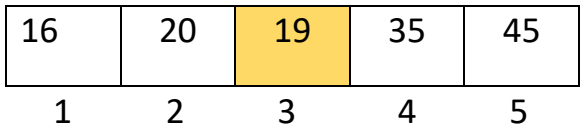
   - **S = {2}** where, S is a singleton set and the only entry is two (user touches two boxes).
     Consider an array,



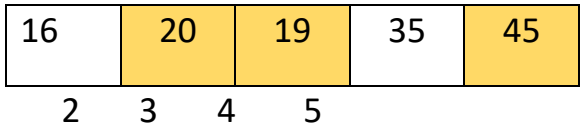
n=2 (two touch)

Let user touch on the box number 3 (index = 3) and box number 5 (index = 5).
We will denote this by (a, b) where a=3 and b=5.
(a, b) is an unordered pair denoting which two boxes are touched.

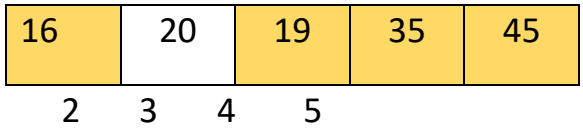
- **S = {1, 3, 4, 5, 6, 7…}** where, S is a set and each entry in S denotes number of boxes touched by the user. Either one or more than equal to three.



n=1 (one touch)



n=3 (three touch)



n=4 (four touch)

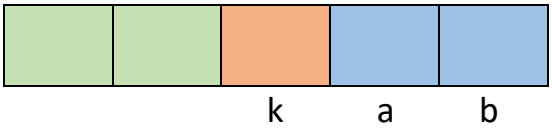
Test cases:

- **Input:** n = 2
  State: Any given state of the game.
  Output: Boxes interchange their position and regain their position if move is not valid (Validation will be checked by other equivalence classes).
- **Input:** n=3
  State: Any given state of the game.
  Output: Nothing happens.

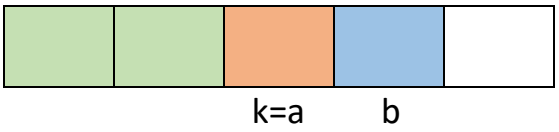
2. **Let sub-list be at index number k (orange box) and all entries before k are sorted (green boxes).**
   Let a, b be the index number of boxes, which user touches (blue boxes).
   So, equivalence class partitioning based on valid or invalid touch (only two touches).

   ## Equivalence classes:

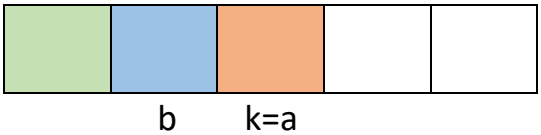
   - **S = { (a, b) | a!=k and b!=k }**
     Set of all unordered pair (a, b) where a and b are not equal to k.
     Invalid touch.



   - **S = { (a, b) | a = k and b > k }**
     Set of all unordered pair (a, b) where a = k and b > k.
     Box with index number k is both orange and blue.
     Valid touch.

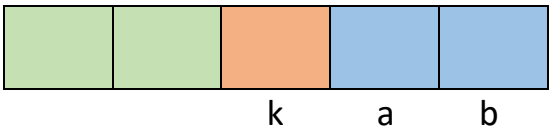


   - **S = { (a, b) | a = k and b < k }**
     Set of all unordered pair (a, b) where a = k and b < k.
     Box with index number k is both orange and blue.
     Invalid touch.



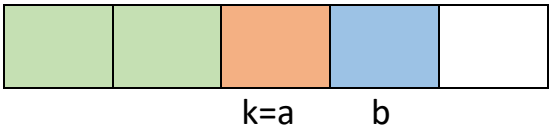
Test cases:

- **Input:** (4, 5) , k=3
  State:



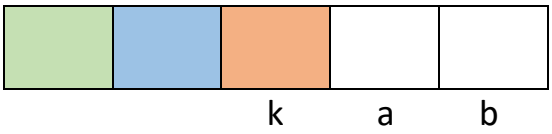
  Output: Boxes interchange their position and regain their position.

- **Input:** (3, 4) , k =4
  State:



  Output: Boxes interchange their position and regain their position if swap is not allowed by algorithm.

- **Input:** (3, 2) , k =2
  State:

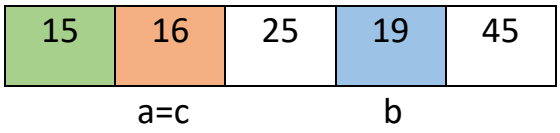


  Output: Boxes interchange their position and regain their position.

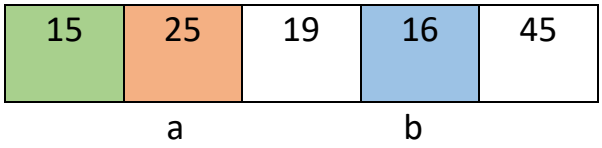
3. **Let sub-list be at index number k (orange box) and all entries before k are sorted (green boxes).**
   Let a, b be the index number of boxes, which user touches (blue boxes) where a = k and b > k (Box with index number k is both orange and blue).
   Let c be index number of box, which has smallest entry among the sub-list starting from index k.
   So, equivalence class partitioning based on valid / invalid swap (move).

Equivalence classes:

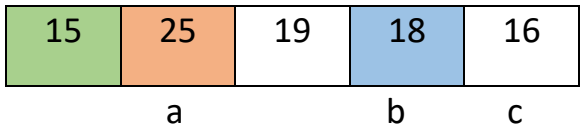
- **S = {(a, b) | a=c}.**
  Smallest entry is in the box with index number a.
  Swapping not possible because smallest element in sub-list [16, 25, 19, 45] is list[a] = 16.



- **S = { (a ,b) | b=c }**
  Swapping possible because smallest element in sub-list [25, 19, 16, 45] is list[b] = 16.
  Smallest entry is in the box with index number b.

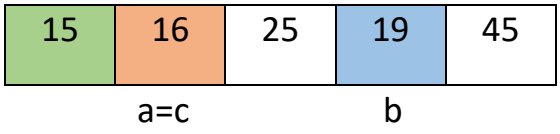


- **S = { (a ,b) | a!= c and b!=c}**
  Swapping not possible because smallest element in sub-list [25, 19, 18, 46] is list[c] = 16.
  Smallest entry is in the box with index number c other than a and b.



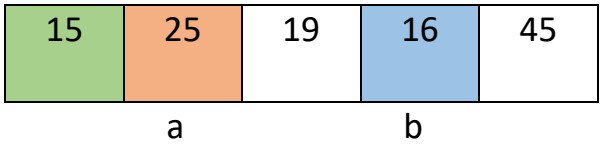
Test cases:

- **Input:** (2, 4) , c = 2
  State:



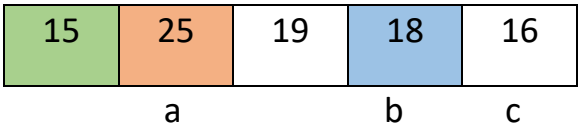
  Output: Boxes interchange their position and regain their position.

- **Input:** (2, 4) , c = 4
  State:



  Output: Boxes interchange their position (This is the only correct move).

- **Input:** (2, 4) , c = 5
  State:



  Output: Boxes interchange their position and regain their position.

2. BUBBLE SORT:

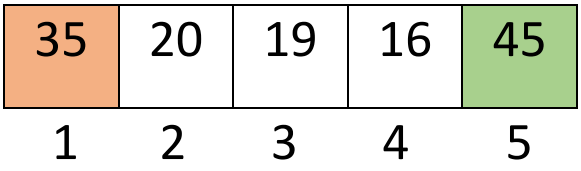
Function Description:

Repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order.

Orange color box is the left box among the pair of adjacent items.

Green color boxes are for sorted element (elements at their correct position in the list).

Consider an array with five entries,

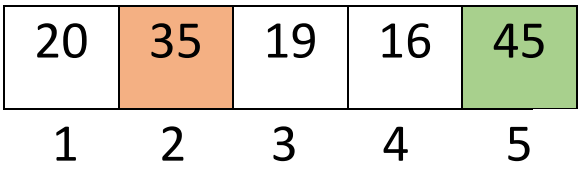


Here, the adjacent pair under consideration is 35 and 20 at index number 1 and 2 respectively (after first **valid** move),

Temp = List [1]

List [1] = List [2]

List [2] = Temp

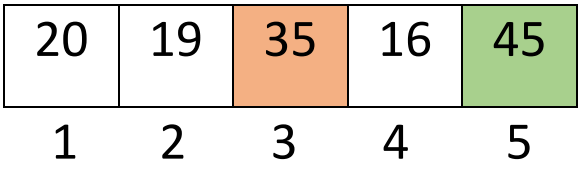


After second **valid** move,

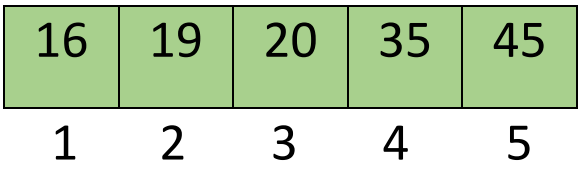
Temp = List [2]

List [2] = List [3]

List [3] = Temp



This process will continue until list is sorted in increasing order correctly



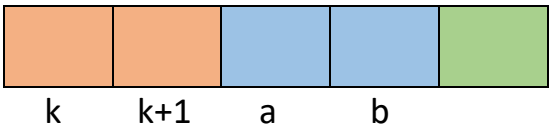
Equivalence class partitioning and Test cases:

1. **System will detect number of touches by user at a particular time instance. Therefore, equivalence class partitioning based on, "how many boxes are touched by user".**
   Equivalence classes and test cases are same as selection sort.

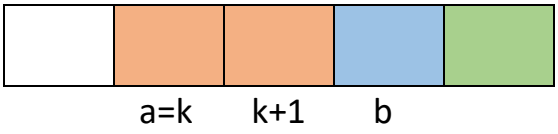
2. **Let index number of adjacent pair of boxes under consideration be k and k+1 (orange color denote this).**
   Let a, b be the index number of boxes, which user touches (blue boxes).
   So, equivalence class partitioning based on valid or invalid touch (only two touches).

   ## Equivalence classes:

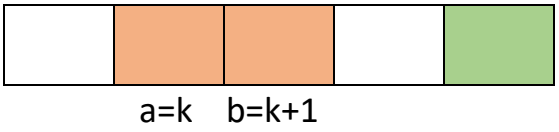
   - **S = { (a, b) | a!=k and b!=k }**
     Set of all unordered pair (a, b) where a and b are not equal to k.
     Invalid touch.



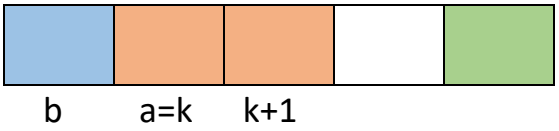
   - **S = { (a, b) | a = k and b > k+1 }**
     Set of all unordered pair (a, b) where a = k and b > k+1.
     Touch on box with index number k and on box with index number greater than k+1.
     Box with index number k is both orange and blue.
     Invalid touch.



- **S = { (a, b) | a = k and b = k+1 }**
  Set of all unordered pair (a, b) where a = k and b < k.
  Touch on box with index number k and on box with index number k+1.
  Box with index number k and k+1 is both orange and blue.
  Invalid touch.

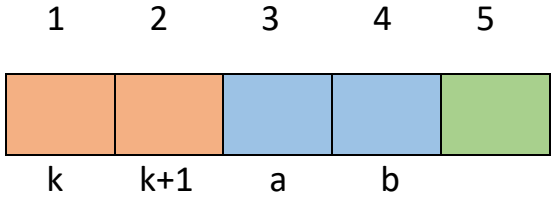


- **S = { (a, b) | a = k and b < k }**
  Set of all unordered pair (a, b) where a = k and b < k.
  Touch on box with index number k and on box with index number smaller than k+1.
  Box with index number k is both orange and blue.
  Invalid touch.



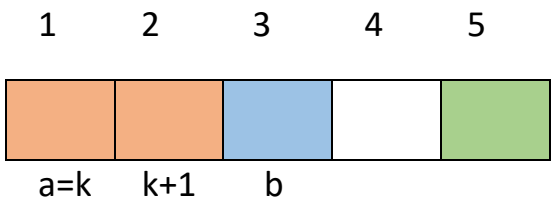
Test cases:

- **Input:** (3, 4) , a=3 , b=4 , k=1
  State:



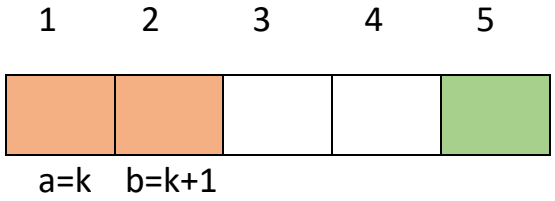
  Output: Boxes interchange their position and regain their position.

- **Input:** (1, 3) , a=1 , b=3 , k = 1
  State:



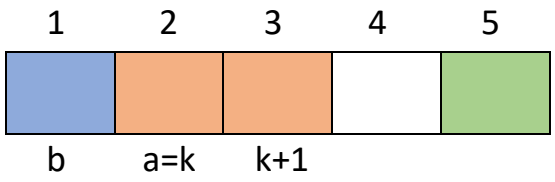
  Output: Boxes interchange their position and regain their position.

- **Input:** (1, 2) , a=1 , b=2 , k = 1
  State:



  Output: Boxes interchange their position and regain their position (if swap is not allowed by algorithm).

- **Input:** (2, 1) , a=2 , b=1 , k = 2
  State:

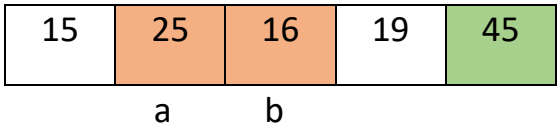


  Output: Boxes interchange their position and regain their position.

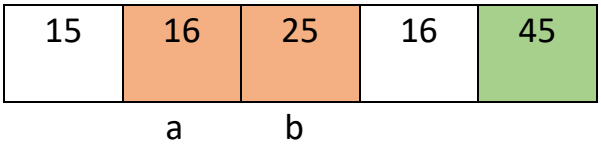
3. **Let index number of adjacent pair of boxes under consideration be a and b (orange color denote this).**
   So, equivalence class partitioning based on valid / invalid swap (move).

   ## Equivalence classes:

   - **S = {(a, b) | b = a+1 and list[a] > list[b] }.**
     Entry in left box is greater than that that of right box.

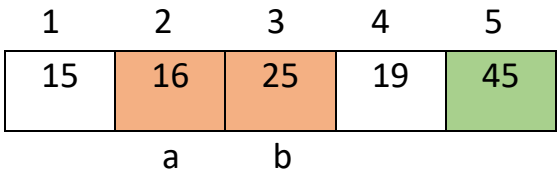


- **S = {(a, b) | b = a+1 and list[a] <= list[b]}.**
  Entry in left box is smaller than or equal to that of right box.



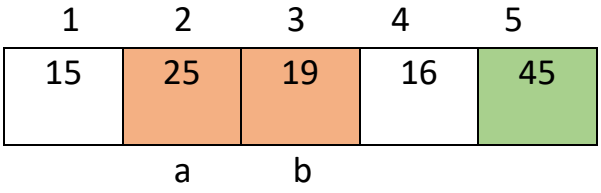
Test cases:

- **Input:** (2, 3) , a = 2 , b = 3 , list [2] >list [3]
  State:



  Output: Boxes interchange their position and regain their position.

- **Input:** (2, 3) , a = 2 , b = 3 , list [2] <= list [3]
  State:



  Output: Boxes interchange their position (This is the only correct move).

3. INSERTION SORT:

Description:

Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list.

Each iteration, insertion sort removes one element from the input data, **finds the location** it belongs within the sorted list, and inserts it there. It repeats until no input elements remain**.**

All elements up to that position **shifts** right.

Orange color box denotes entry in that box is consumed to insert it in sorted list.

Green color box denotes the current sorted sub-list.

Consider an array with five entries,

25	35	45	30	18

Here, sorted sub-list is [25, 35, 45] and 30 at index number 4 is consumed to be inserted in sub-list.

After first **valid** move,

25	30	35	45	18

Here, sorted sub-list is [25, 30, 35, 45] and 18 at index number 5 is consumed to be inserted in sub-list.

After second **valid** move,

15	25	19	16	45

Equivalence class partitioning and test cases:

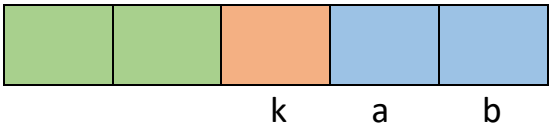
1. **System will detect number of touches by user at a particular time instance. Therefore, equivalence class partitioning based on, "how many boxes are touched by user".**

   Equivalence classes and test cases are same as selection sort.

2. **Let index number of box to be consumed for inserting it in the sub-list be k (orange color denote this).**
   Let a, b be the index number of boxes, which user touches (blue boxes).
   So, equivalence class partitioning based on valid or invalid touch (only two touches).

   ## Equivalence classes:

   - **S = {(a, b) | a! =k and b! =k}**
     Set of all unordered pair (a, b) where a and b are not equal to k.
     Invalid touch.



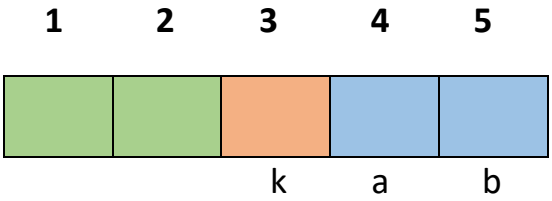
   - **S = {(a, b) | a = k and b > k}**
     Set of all unordered pair (a, b) where a = k and b > k+1.
     Touch on box with index number k and on box with index number greater than k.
     Box with index number k is both orange and blue.
     Invalid touch.

a=k b

- **S = {(a, b) | a = k and b < k}**
  Set of all unordered pair (a, b) where a = k and b < k.
  Touch on box with index number k and on box with index number smaller than k+1.
  Box with index number k is both orange and blue.
  Invalid touch.

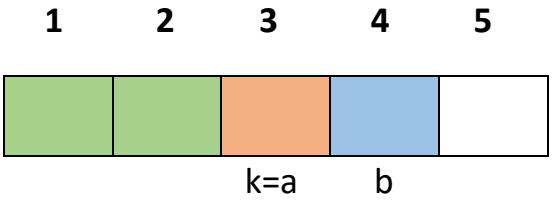
Test cases:

- **Input:** (4, 5), a = 4, b = 5, k=3
  State:

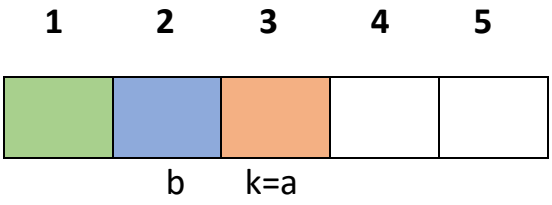


  Output: Boxes interchange their position and regain their position.

- **Input:** (3, 4), a = 3, b = 4, k=3
  State:



  Output: Boxes interchange their position and regain their position.
- **Input:** (3, 2), a = 3, k=3
  State:

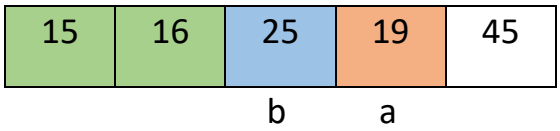


Output: Boxes interchange their position if insertion algorithm allows it and regain their otherwise.

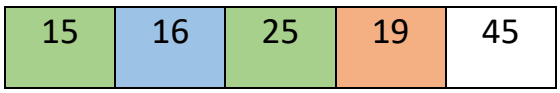
3. **Let index number of box to be consumed for inserting it in the sub-list be k (orange color denote this).**
   Let a, b be the index number of boxes, which user touches (blue boxes).
   So, equivalence class partitioning based on valid/invalid swap (only two touches) i.e. weather box with index number b be inserted just before box with index number a (validity check by insertion algorithm).

   ## Equivalence classes:

   - **S = {(a, b) | list[b] > list[a] and list[c] < list[a] for all c < b}.**
     Valid swap because all entries in green boxes before blue box is smaller than entry at orange box and entry at blue box is larger than entry at orange box.

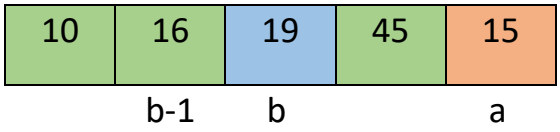


   - **S = {(a, b) | list[b] < list[a] or list[b] > list[a] and list[c] < list[a] for some c < b}.**
     Invalid swap because box with index a is not inserted at its proper position.



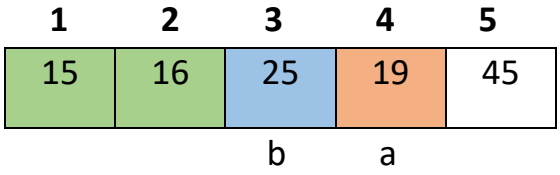
b a

- **S = {(a, b) | list[b] > list[a] and list[b-1] > list[a] }.**
  Invalid swap because box with index a is not inserted at its proper position.



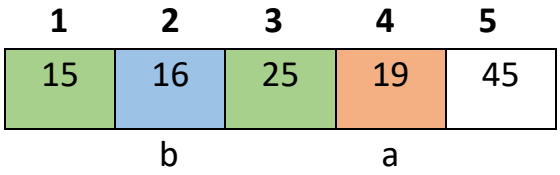
Test cases:

- **Input:** (4, 5), a = 4, b = 5, list [2] <list [4], list [3] > list [4]
  State:



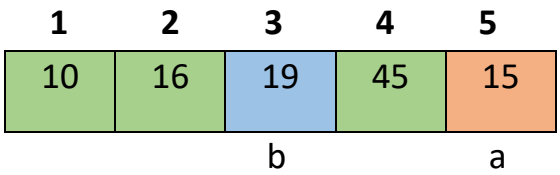
  Output: Boxes interchange their position (This is the only valid move).

- **Input:** (4, 2), a = 4, b = 2, list [2] < list [4]
  State:



  Output: Boxes interchange their position and regain their position.

- **Input:** (5, 3), a = 5, b = 3, list [3] > list [5] but list [2] > list [5]
  State:



  Output: Boxes interchange their position and regain their position.

4. HIGHSCORE:

Description:

Highest score till date on running app will be displayed on the screen when someone plays the game.

Equivalence class partitioning:

Consider highest score be H and the score of the player after he finishes the game be S. Denote this ordered pair (H, S).

- **(H, S) and he loses the game.**
  As long as player loses the game, his current score doesn't matter because highest score will update only when he wins the game.
- **(H, S) where H > S and he wins the game.**
  This time score will update since he wins the game and his score is greater than highest score till now.
- **(H, S) where H <= S and he wins the game.**
  Although player wins the game, highest score will not update because his current score is less than highest score.

Test cases:

- **Input:** (20, 40) where H=20 and S=40
  State: Player loses
  Output: Highest score will not update.

- **Input:** (20, 40) where H=20 and S=40
  State: Player wins
  Output: Highest will update and will be equal to 40 since S >H.

- . **Input:** (40, 20) where H=40 and S=20
  State: Player wins.
  Output: Highest will not update since H > S

5. SCOREBOARD:

Description:

Current score of the player will update on the basis of validation of move.

+ 10 if a move is valid.

- 10 if a move is invalid.

Equivalence class partitioning:

Let a and b are the index number of box which user touches, call this unordered pair (a, b).

- (a, b) is invalid move.
- (a, b) is valid move.

Test cases:

- **Input:** Valid move.
  State: Any state of the game.
  Output: Score will increase by 10.

- **Input:** Invalid move.
  State: Any state of the game.
  Output: Highest will decrease by 10.

6. MUSIC CONTROL:

Description:

A soundtrack is included in the game that constantly plays in the background and can be turn on/off as required.

Equivalence class partitioning:

- Music on
- Music off

Test cases:

- **Input:** Tap.
  State: Music is off.
  Output: Music will play in the background.

- **Input:** Tap.
  State: Music is playing.
  Output: Music will turn off.

7. GAME WON/GAME LOST:

Description:

If three consecutive invalid moves are there, then the game is lost.

On the other hand, if player manages to sort the list without playing three consecutive wrong moves then he wins the game.

Equivalence class partitioning:

- Three wrong moves by the player.
- Player correctly sorts the list without playing three consecutive wrong moves.

Test cases:

- **Input:** Three consecutive invalid moves.
  State: Any state of the game.
  Output: Player loses the game.

- **Input:** A valid move after two consecutive invalid moves.
  State: Any state of the game.
  Output: Scoreboard will update and game proceeds.