FOP – Searching and Sorting

Searching and Sorting

Objectives:

- Understand the basic searching algorithms and sorting algorithms
- Introduce the concept of complexity analysis (informally)
- Implement the searching and sorting algorithms using arrays.

Reference:

- III.8. ARRAY SORTING
  - Lesson III.8.2 Bubble Sort, III.8.3 Selection Sort, III.8.4 Insertion Sort

Searching: Outline (1/2)

1. String Recap Exercise
2. Overall Introduction
3. Introduction to Searching
4. Linear Search
   - Demo #1
   - Performance
5. Exercise #1: Compatible Teammate
6. Binary Search

Sorting: Outline (2/2)

7. Introduction to Sorting
8. Selection Sort
   - Demo #2
   - Performance
9. Exercise #2: Points and Lines
10. Bubble Sort
    - Demo #3
    - Performance
11. Insertion Sort
12. Animated Sorting Algorithms
13. Exercise #3: Module Sorting (take-home)

1. String recap exercise: Arrow Program (1/2)

- Write a program to randomly select a student to answer question.
- The program reads in a list of names and use rand() to randomly select one of the names.
- When a name is selected, the program will print out the first name, followed by the last name. For example, if the selected name is Gaju Bryan. The program will print:

```
Gaju Bryan, would you please answer the question?
```

- You may assume that each name contains at most 30 characters, and there are at most 12 names.

1. String recap exercise.: Arrow Program (2/2)

- A sample run:

```
Enter names:
Hirwa Seth
Karabo Mugisha Aline
Gaju Bryan
Kalisa Nina
Neza Cyuzuzo Hagai
Mahoro Peace

Name selected:
Gaju Bryan, would you please answer the question?
```

```
printf("Enter names (terminate with ctrl-d):\n");
  while (fgets(nameList[i], NAME_LENGTH+1, stdin) != NULL) {
     if (i++ >= GRP_SIZE)
        break;
  }
```

```
  srand(time(NULL));
  index = (rand() / (double) RAND_MAX) * (i - 1);

  //Find the last and first name
  strcpy(lastname, strtok(nameList[index], " "));
  strcpy(firstname, strtok(NULL, " "));

  printf("Name selected:\n");
  printf("%s %s, would you please answer the question?\n“,
         firstname, lastname);
```

2. Overall Introduction

- You have accumulated quite a bit of basic programming experience by now.
- In this session we will study some simple yet useful classical algorithms which find their place in many CS applications
  - Searching for some data amid very large collection of data
  - Sorting very large collection of data according to some order
  - How large is the collection of data that we are considering?
- We will begin with an algorithm (idea), and show how the algorithm is transformed into a C program (implementation).

3. Introduction to Searching (1/2)

- Searching is a common task that we carry out without much thought everyday.
  - Searching for a location in a map
  - Searching for the contact of a particular person
  - Searching for a nice picture for your project report
  - Searching for an online course required in your course
  - Searching for a book required in your course.
- In this session, you will learn how to search for an item (sometimes called a search key) in an array.

3. Introduction to Searching (2/2)

- Problem statement:

  Given a list (collection of data) and a search key X, return the position of X in the list if it exists.

  For simplicity, we shall assume there are no duplicate values in the list.

- Our learning objectives:
  - Two methods of finding an item with the search key X
    - Linear/Sequential Search
    - Binary Search
  - Quantitative assessment of the quality of the method

3. Introduction to Searching (2/2)

- Problem statement:

  Given a list (collection of data) and a search key X, return the position of X in the list if it exists.

  For simplicity, we shall assume there are no duplicate values in the list.

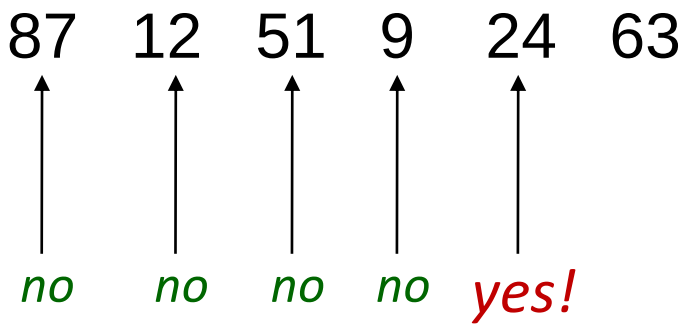
- We will count the number of comparisons the algorithms make to analyze their performance.
  - The ideal searching algorithm will make *the least possible number of comparisons* to locate the desired data.
  - We will introduce worst-case scenario.

4. Linear Search (1/3)

- Also known as Sequential Search
- Idea: Search the list from one end to the other end in linear progression (from starting to end or from end to the start).
- Algorithm

```
// Search for key in list A with n items
linear_search(A, n, key)
{
    for i = 0 to n-1
        if A_i is key then report i
}
```

Example: Search for 24
in this list



- Note: If the array is only partially filled, the search stops when the last meaningful value has been checked

4. Demo #1: Linear Search (2/3)

- If the list is an array, how would you implement the Linear Search algorithm?

```
// To search for key in arr using linear search
// Return index if found; otherwise return -1
int linearSearch(int arr[], int size, int key) {
    int i;

    for (i=0; i<size; i++)
        if (key == arr[i])
            return i;
    return -1;
}
```

- Question: What if array contains duplicate values of the key?

4. Linear Search: Performance Analysis (3/3)

- We use the number of comparisons here as a rough measurement.
- Analysis is usually done for best case, average case, and worst case. We will focus on the worst case.
- For an array with *n* elements, in the worst case,
  - What is the number of comparisons in our linear search algorithm?

    n comparisons

  - Under what circumstances do we encounter the worst case?

    (a) Not found
    (b) Found at last element

5. Exercise #1: Compatible Teammate (1/4)

- Given three arrays:
  1. The first array contains the players' names.
  2. The second array stores the players' ages corresponding to the entries in the first array.
  3. The third array contains the gender information for each player corresponding to the entries in the first array.
- Your task is to accept as input a player's name and find a compatible teammate for this player – a teammate who is of the same age but opposite sex.
- Analysis:

Inputs:

```
char player_name[STR_LENGTH+1];
char names[MAX_PLAYER][STR_LENGTH+1];
int ages[MAX_PLAYER];
char genders[MAX_PLAYER];
```

5. Exercise #1: Compatible Teammate (2/4)

- Sample run:

```
Enter 5 players' info:
Hirwa 23 M
Gaju 21 M
Karabo 19 F
Neza 21 F
Mahoro 22 M
The list of players are:
Name     Age      Gender
Hirwa    23       M
Gaju     21       M
Karabo   19       F
Neza     21       F
Mahoro   22       M
```

```
Enter a player's name: Gaju
Gaju's compatible teammate is Neza.
```

or

```
Enter a player's name: Hirwa
Sorry, we cannot find a teammate for
Hirwa!
```

or

```
Enter a player's name: Keza
No such player Keza.
```

- Use the the incomplete program "TeamMate" from next slide.
- Complete the search_teammate() function.

5. Exercise #1: Compatible Teammate (3/4)

```c
int main(void) {
     char names[MAX_PLAYER][STR_LENGTH+1];
     int  ages[MAX_PLAYER];
     char genders[MAX_PLAYER];
     char player_name[STR_LENGTH+1];
     int  i, result;

     printf("Enter %d players' info:\n", MAX_PLAYER);
     for (i=0; i<MAX_PLAYER; i++)
          scanf("%s %d %c", names[i], &ages[i], &genders[i]);

     printf("The list of players are:\n");
     printf("Name\tAge\tGender\n");
     for (i=0; i<MAX_PLAYER; i++)
          printf("%s\t%d\t%c\n", names[i], ages[i], genders[i]);

     printf("Enter a player's name: ");
     scanf("%s", player_name);

     result = search_teammate(names, ages, genders, MAX_PLAYER, player_name);
     if (result == -2)
          printf("No such player %s.\n", player_name);
     else if (result == -1)
          printf("Sorry, we cannot find a teammate for %s!\n", player_name);
     else
          printf("%s's compatible teammate is %s.\n", player_name, names[result]);
     return 0;
}
```

5. Exercise #1: Compatible Teammate (4/4)

```
// Search for a player's compatible teammate
// Return index in array if compatible teammate found, or
//        -1 if compatible teammate not found, or
//        -2 if player not found
int search_teammate(char names[][STR_LENGTH+1], int ages[], char genders[],
                    int size, char player_name[]) {
     int i, player_index = -1;

     // First, check that the player_name appears in the names array
     for (i=0; i<size; i++) {
          if (strcmp(player_name, names[i]) == 0)
               player_index = i;
     }

     if (player_index == -1)
          return -2;  // no such student in array

     for (i=0; i<size; i++) {
          if ((ages[i] == ages[player_index]) &&
                    (genders[i] != genders[player_index]))
               return i;
     }

     return -1; // cannot find compatible teammate
}
```

6. Binary Search (1/6)

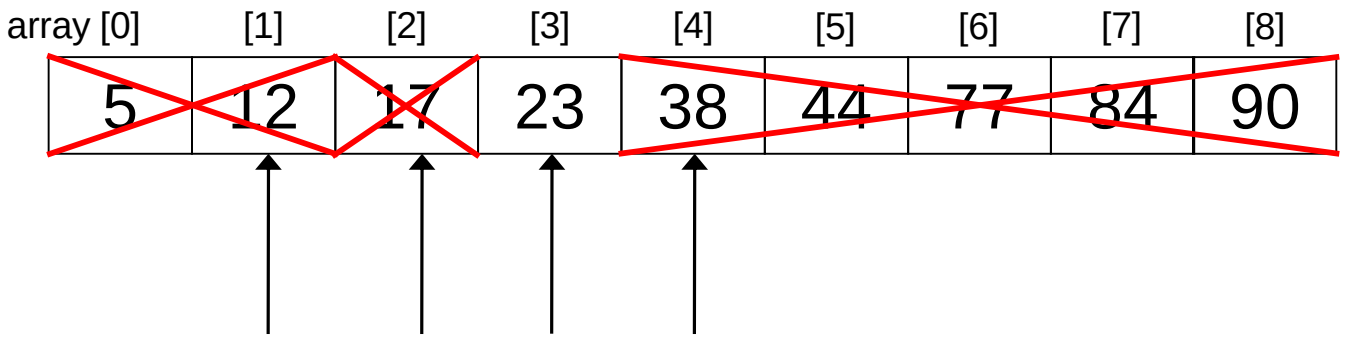
- The idea is simple and fantastic, but when applied on the searching problem, it has this pre-condition that the list must be sorted before-hand.
- How the data is organized (in this case, sorted) usually affects how we choose/design an algorithm to access them.
- In other words, sometimes we seek out new way to organize the data so that we can process them more efficiency.

6. Binary Search (2/6)

- The Binary Search algorithm
  - Look for the key in the middle position of the list. Either of the following 2 cases happens:
    - If the key is smaller than the middle element, then “discard” the right half of the list and repeat the process.
    - If the key is greater than the middle element, then “discard” the left half of the list and repeat the process.
  - Terminating condition: when the key is found, or when all elements have been “discarded”.

6. Binary Search (3/6)

- Example: Search key = 23



1. low = 0, high = 8, mid = (0+8)/2 = 4
2. low = 0, high = 3, mid = (0+3)/2 = 1
3. low = 2, high = 3, mid = (2+3)/2 = 2
4. low = 3, high = 3, mid = (3+3)/2 = 3

Found!
Return 3

6. Binary Search (4/6)

- In binary search, each step eliminates the problem size (array size) by half.
  - The problem size gets reduced to 1 very quickly! (see next slide)
- This is a simple yet powerful strategy, of halving the solution space in each step
- Such strategy, a special case of divide-and-conquer paradigm, can be naturally implemented using recursion (a topic we already covered in the past)
- But for this session, we will stick to iteration (loop)

6. Binary Search (5/6): Performance

- Worst-case analysis

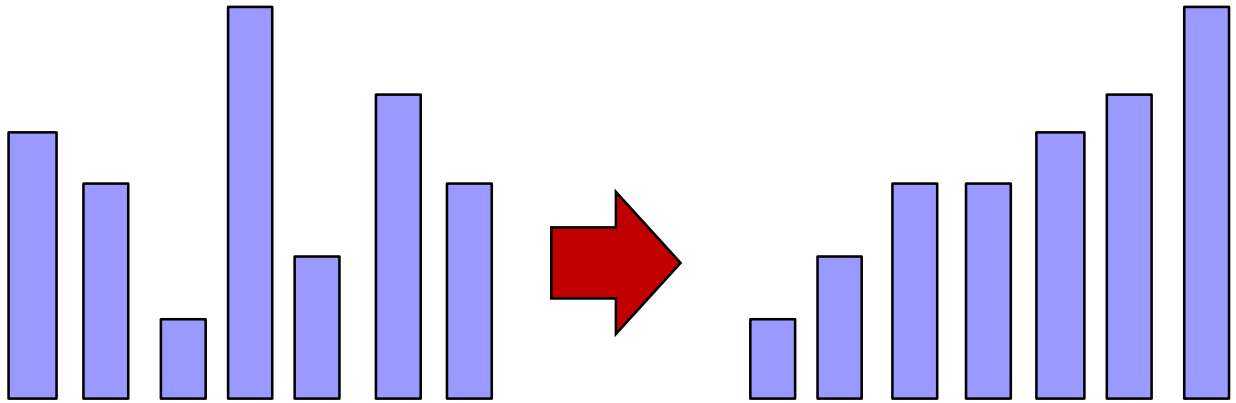
Array size *n*	Linear Search (*n* comparisons)	Binary search ($\log_2 n$ comparisons)
100	100	≈7
1,000	1,000	≈10
10,000	10,000	≈14
100,000	100,000	≈17
1,000,000	1,000,000	≈20
10^9	10^9	≈30

6. Binary Search (6/6): Code

```
// To search for key in sorted arr using binary search
// Return index if found; otherwise return -1
int binarySearch(int arr[], int size, int key) {

    int low = 0, high = size - 1, mid = (low + high)/2;

    while ((low <= high) && (arr[mid] != key)) {
        if (key < arr[mid])
            high = mid - 1;
        else
            low = mid + 1;
        mid = (low + high)/2;
    }
    if (low > high) return -1;
    else            return mid;

}
```

7. Introduction to Sorting (1/2)

- Sorting is any process of arranging items in some sequence and/or in different sets.
- Sorting is important because once a set of items is sorted, many problems (such as searching) become easy.
  - Searching can be speeded up.
  - Determining whether the items in a set are all unique.
  - Finding the median item in the set.
  - etc.



7. Introduction to Sorting (2/2)

- Problem statement:

  Given a list of *n* items, arrange the items into ascending order.

- We will implement the list as an integer array.
- We will introduce three basic sort algorithms.
- We will count the number of comparisons the algorithms make to analyze their performance.
  - The ideal sorting algorithm will make the least possible number of comparisons to arrange data in a designated order.
- We will compare the algorithms by analyzing their worst-case performance.

8. Selection Sort (1/5)

- Selection Sort algorithm

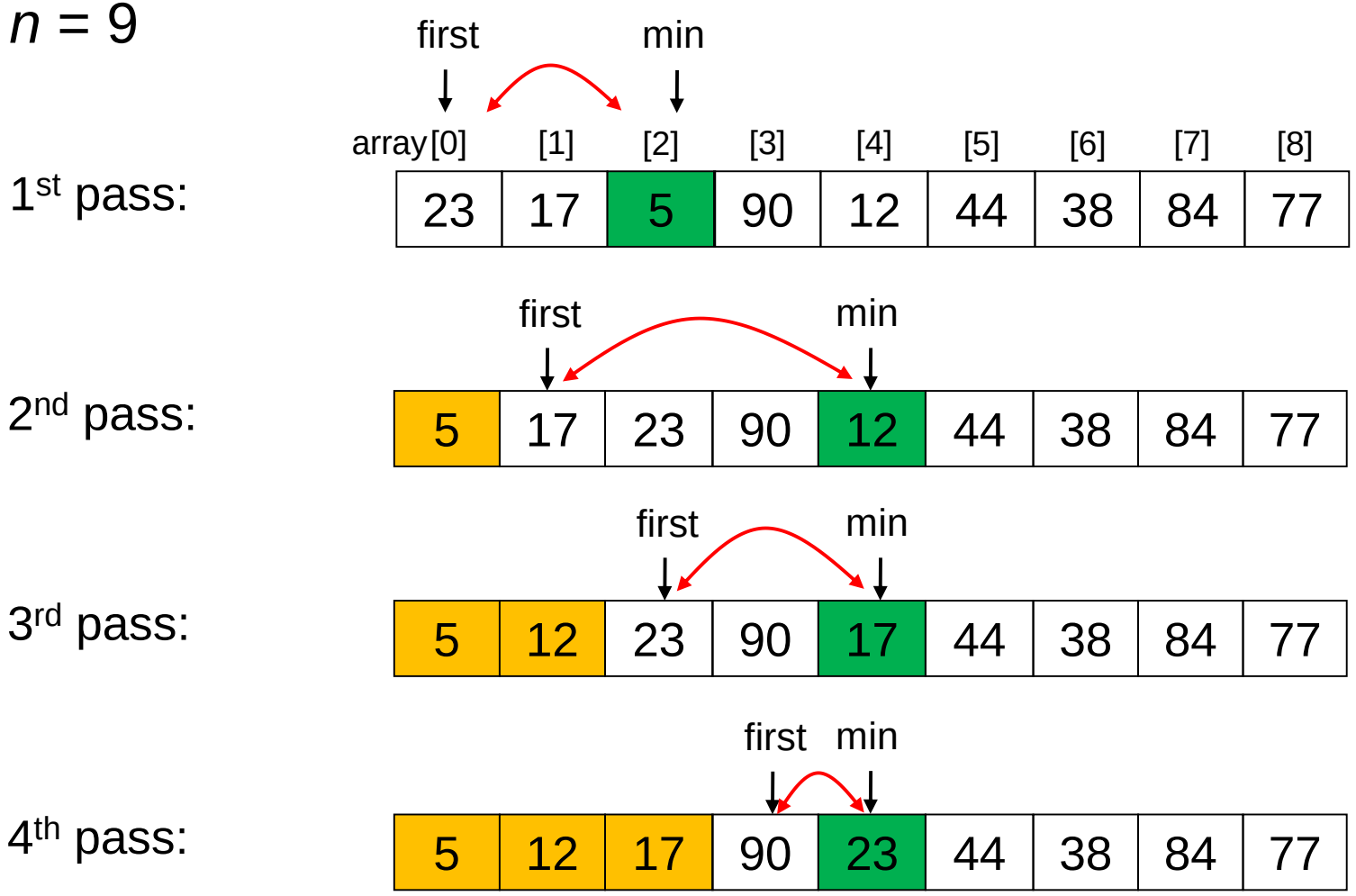
  Step 1: Find the smallest element in the list (find_min)

  Step 2: Swap this smallest element with the element in the first position. (Now, the smallest element is in the right place.)

  Step 3: Repeat steps 1 and 2 with the list having one fewer element (i.e. the smallest element just found and placed is “discarded” from further processing).

8. Selection Sort (2/5)

$n = 9$



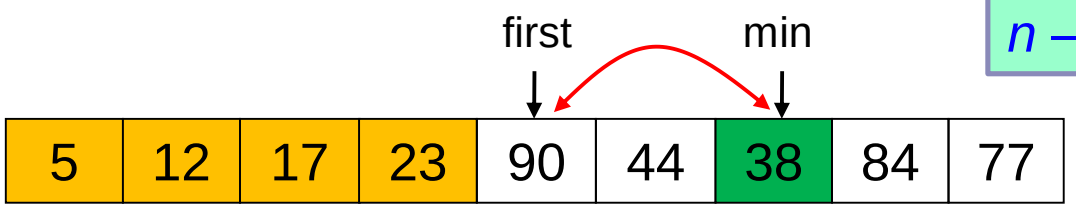
8. Selection Sort (3/5)

Q: How many passes for an array with *n* elements?

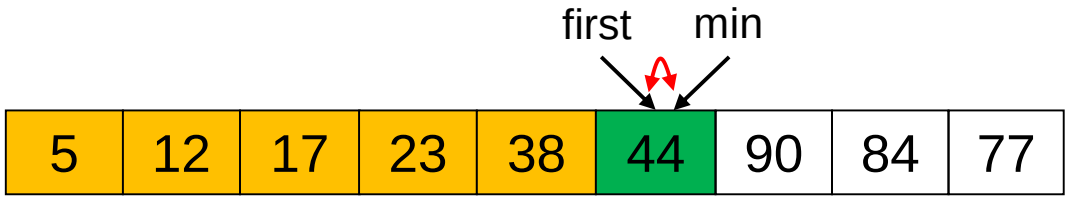
n – 1 passes

n = 9

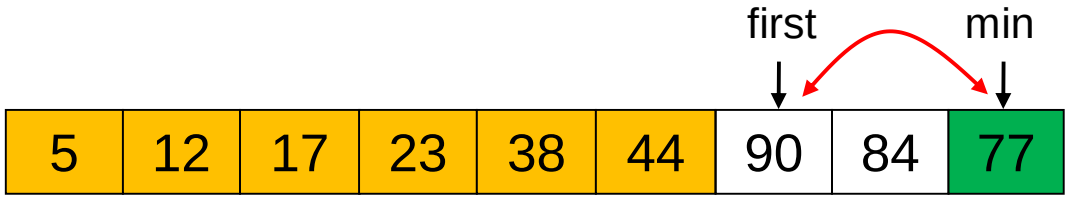
5th pass:



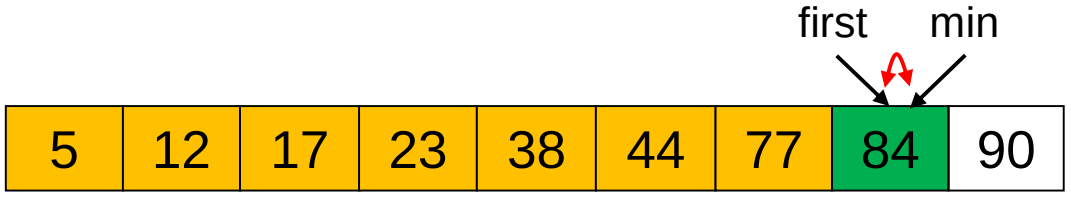
6th pass:



7th pass:



8th pass:



Final array:

8. Demo #2: Selection Sort (4/5)

```
// To sort arr in increasing order
void selectionSort(int arr[], int size) {
    int i, start, min_index, temp;

    for (start = 0; start < size-1; start++) {
       // each iteration of the for loop is one pass

       // find the index of minimum element
       min_index = start;
       for (i = start+1; i < size; i++)
           if (arr[i] < arr[min_index])
              min_index = i;

       // swap minimum element with element at start index
       temp = arr[start];
       arr[start] = arr[min_index];
       arr[min_index] = temp;
    }
}
```

8. Selection Sort: Performance (5/5)

- We choose the number of comparisons as our basis of analysis.
- Comparisons of array elements occur in the inner loop, where the minimum element is determined.
- Assuming an array with *n* elements. Table below shows the number of comparisons for each pass.
- The total number of comparisons is calculated in the formula below.
- Such an algorithm is call an n^2 algorithm, or quadratic algorithm, in terms of running time complexity.

Pass	#comparisons
1	$n - 1$
2	$n - 2$
3	$n - 3$
…	…
$n - 1$	1

$$\sum_{i=1}^{n-1} i = \frac{(n-1)(n)}{2} = \frac{n^2 - n}{2} \cong n^2$$

9. Exercise #2: Points and Lines (1/6)

- Problem: You are given a list of points on a 2-dimensional plane, each point represented by its integer x- and y-coordinates. You are to sort the points in ascending order of their x-coordinates, and for those with the same x-coordinate, in ascending order of their y-coordinates.
- Two arrays are used to store the points: array x for their x-coordinates, and array y for their y-coordinates. x[i] and y[i] refer to the point i.
- You may assume that there are at most 20 points and no two points are the same.
- Do the sorting by calling Selection Sort only once. How do you adapt the Selection Sort code for this problem?
- You are given an incomplete program Points.c fdfdfdfdsfsdfdfsdf

9. Exercise #2: Points and Lines (2/6)

- In the preceding section, the comparison is done using a simple relational operation:

```
for (i = start_index+1; i < size; i++)
    if (arr[i] < arr[min_index])
        min_index = i;
```

- What if the problem deals with more complex data, like this exercise?
- The simple relational operation above would have to be replaced by a more complex one. Or you could call another function to do it. For example, for this exercise:

```
for (i = start_index+1; i < size; i++)
    if ( lessThan(x, y, i, min_index) )
        min_index = i;
```

9. Exercise #2: Points and Lines (3/6)

- Here's the incomplete lessThan() function:

```
// Returns 1 if point at index p is "less than" point at index q;
// otherwise returns 0.
// Point at index p is "less than" point at index q if the former
// has a smaller x-coordinate, or if their x-coordinates are the
// same, then the former has a smaller y-coordinate.
int lessThan(int x[], int y[], int p, int q) {


   return (x[p] < x[q])
            || ( (x[p] == x[q]) && (y[p] < y[q]) );




}
```

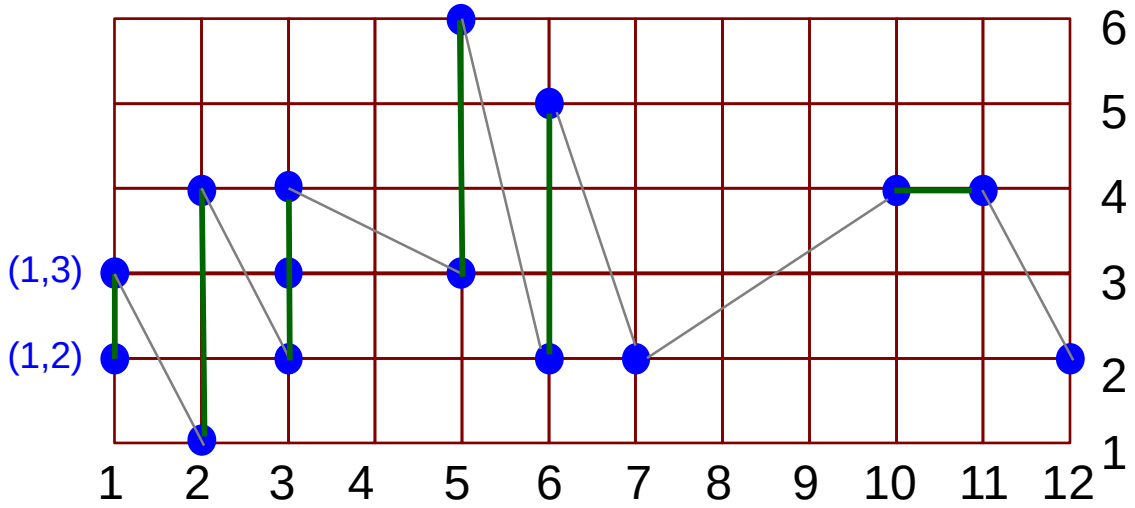
9. Exercise #2: Points and Lines (4/6)

- Here's the incomplete sortPoints() function:

```
// Sort points in ascending order of x-, and then y-coordinates
void sortPoints(int x[], int y, int size) {
    int i, start, min_index, temp;

    for (start = 0; start < size-1; start++) {
        // find the index of minimum element
        min_index = start;
        for (i = start+1; i < size; i++)
            // check if point at index i is "less than" point at min_index
            if ( lessThan(x, y, i, min_index) )
                min_index = i;

        // swap minimum element with element at start index
        temp = x[start]; x[start] = x[min_index]; x[min_index]=temp;
        temp = y[start]; y[start] = y[min_index]; y[min_index]=temp;
    }


}
```

9. Exercise #2: Points and Lines (5/6)

- After sorting the points, imagine that you trace the points in their order in the sorted array. Write a function traceLines() to compute the sum of the lengths of those lines that are horizontal or vertical.
- For example, after sorting, here are the points: (1,2), (1,3), (2,1), (2,4), (3,2), (3,3), (3,4), (5,3), (5,6), (6,2), (6,5), (7,2), (10,4), (11,4), (12,2). The vertical and horizontal lines are marked in green.



- Sum of lengths of horizontal and vertical lines = 1 + 3 + 2 + 3 + 3 + 1 = 13

9. Exercise #2: Points and Lines (6/6)

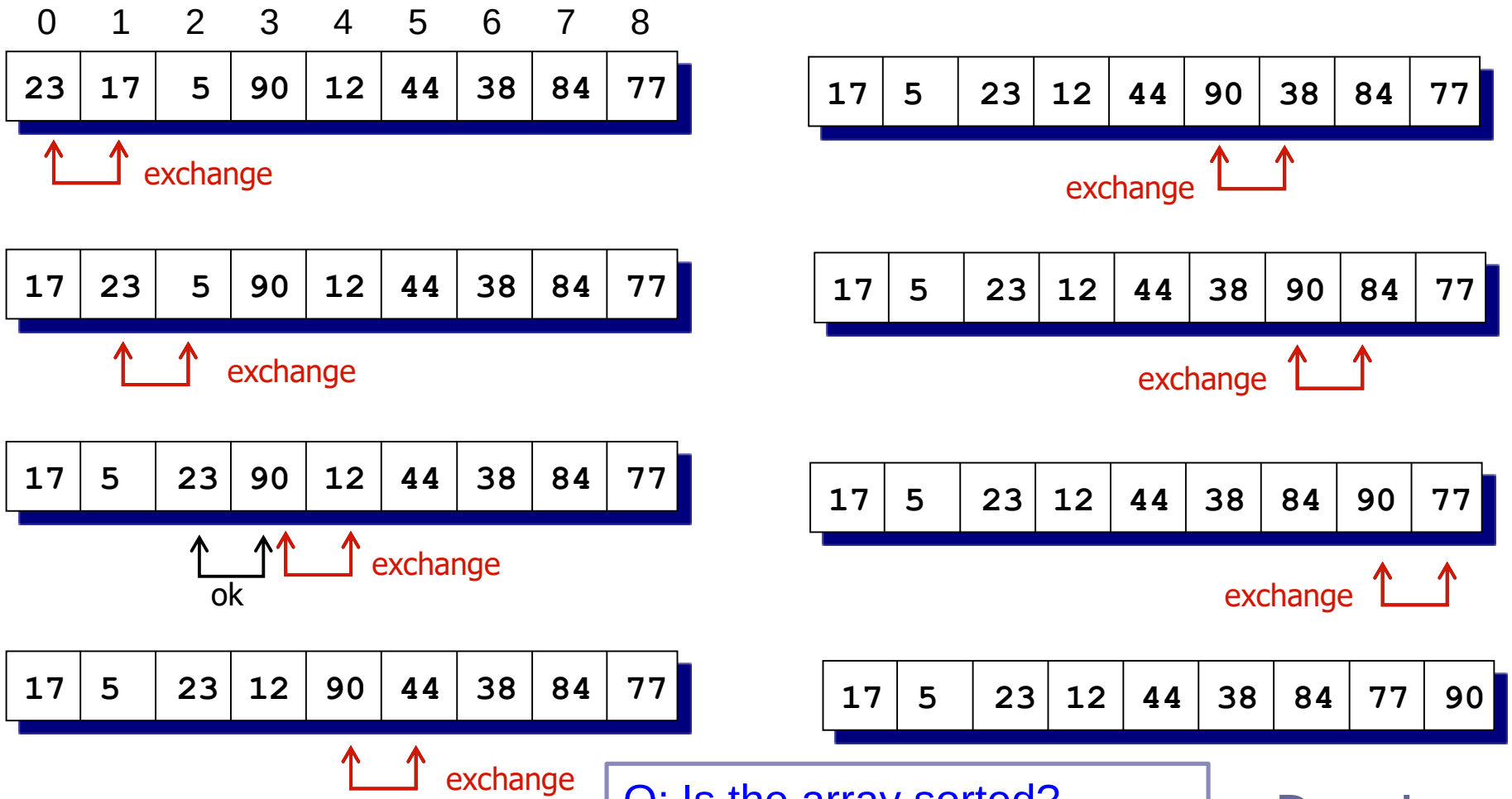
- Sample run:

```
Enter number of points: 15
Enter x- and y-coordinates of 15 points:
5 3
2 4
 :
2 1
After sort:
Point # 0: (1,2)
Point # 1: (1,3)
 :
Point #14: (12,2)
Sum of lengths of vertical and horizontal lines = 13
```

10. Bubble Sort (1/5)

- Bubble sort makes one exchange at the end of each pass.
- What if we make more than one exchange during each pass?
- The key idea of the bubble sort is to make pairwise comparisons and exchange the positions of the pair if they are in the wrong order.

10. One Pass of Bubble Sort (2/5)



Q: Is the array sorted?
Q: What have we achieved?

Done!

Bubble Sort: Example with all possible iterations (3/5)

- First Iteration:
  - [5, 3], 4, 9, 2 → [3, 5], 4, 9, 2
  - 3, [5, 4], 9, 2 → 3, [4, 5], 9, 2
  - 3, 4, [5, 9], 2 → 3, 4, [5, 9], 2
  - 3, 4, 5, [9, 2] → 3, 4, 5, [2, 9]
- Second Iteration:
  - [3, 4], 5, 2, **9** → [3, 4], 5, 2, **9**
  - 3, [4, 5], 2, **9** → 3, [4, 5], 2, **9**
  - 3, 4, [5, 2], **9** → 3, 4, [2, 5], **9**
- Third Iteration:
  - [3, 4], 2, **5, 9** → [3, 4], 2, **5, 9**
  - 3, [4, 2], **5, 9** → 3, [2, 4], **5, 9**
- Fourth Iteration:
  - [3, 2], **4, 5, 9** → [2, 3], **4, 5, 9**

10. Demo #3: Bubble Sort (4/5)

```
// To sort arr in increasing order
void bubbleSort(int arr[], int size) {
   int i, limit, temp;

   for (limit = size-2; limit >= 0; limit--) {
   //limit is where the inner loop variable i should end

      for (i=0; i<=limit; i++) {
         if (arr[i] > arr[i+1]) { //swap arr[i] with arr[i+1]
            temp = arr[i];
            arr[i] = arr[i+1];
            arr[i+1] = temp;
         }
      }
   }
}
```

10. Bubble Sort Performance (5/5)

- Bubble sort, like selection sort, requires $n - 1$ passes for an array with n elements.
- The comparisons occur in the inner loop. The number of comparisons in each pass is given in the table below.
- The total number of comparisons is calculated in the formula below.
- Like selection sort, bubble sort is also an n^2 algorithm, or quadratic algorithm, in terms of running time complexity.

Pass	#comparisons
1	$n - 1$
2	$n - 2$
3	$n - 3$
…	…
$n - 1$	1

$$\sum_{i=1}^{n-1} i = \frac{(n-1)(n)}{2} = \frac{n^2 - n}{2} \cong n^2$$

11. Insertion Sort (1/2)

- Basic Idea:
  - Keeping expanding the sorted portion by one
  - Insert the next element into the right position in the sorted portion
- Algorithm:
  1. Start with one element [is it sorted?] – sorted portion
  2. While the sorted portion is not the entire array
     1. Find the right position in the sorted portion for the next element
     2. Insert the element
     3. If necessary, move the other elements down
     4. Expand the sorted portion by one

11. Insertion Sort: An example (2/2)

- First iteration
  - Before: **[5]**, 3, 4, 9, 2
  - After: **[**3, **5]**, 4, 9, 2
- Second iteration
  - Before: **[3, 5]**, 4, 9, 2
  - After: **[3**, 4, **5]**, 9, 2
- Third iteration
  - Before: **[3, 4, 5]**, 9, 2
  - After: **[3, 4, 5**, 9**]**, 2
- Fourth iteration
  - Before: **[3, 4, 5, 9]**, 2
  - After: **[**2, **3, 4, 5, 9]**

12. Animated Searching Algorithms

- You can find animated searching algorithms on the Internet at this link
  - https://www.cs.usfca.edu/~galles/visualization/Search.html
- There are also folk dances based on searching
  - Linear search with FLAMENCO dance http://www.youtube.com/watch?v=Ns4TPTC8whw
  - Binary search with FLAMENCO dance https://www.youtube.com/watch?v=iP897Z5Nerk

12. Animated Sorting Algorithms

- There are a number of animated sorting algorithms on the Internet
- Here is the site:
  - http://www.sorting-algorithms.com/
  - https://www.youtube.com/watch?v=GIvjJwzrHBU
  - https://www.youtube.com/watch?v=kPRA0W1kECg
- There are also folk dances based on sorting from YouTube video.
  - Selection sort with Gypsy folk dance http://www.youtube.com/watch?v=Ns4TPTC8whw
  - Bubble-sort with Hungarian folk dance http://www.youtube.com/watch?v=lyZQPjUT5B4
  - Insertion-sort with Romanian folk dance https://www.youtube.com/watch?v=ROalU379l3U
  - Quick-sort with Hungarian (Kükullőmenti legényes) folk dance (not covered) https://www.youtube.com/watch?v=ywWBy6J5gz8
- And more others (Merge-sort, Quick sort (the best), Shell-sort, Heap sort, …)

13. Exercise #3: Module Sorting (1/2)

- Given two arrays: one containing the module codes, and the other containing the number of students enrolled in the modules. Sort the modules in ascending order of student enrolment, using Selection Sort, Bubble Sort, or Insertion Sort.
- You may assume that there are at most 10 modules and a module code is at most 7 characters long.

13. Exercise #3: Module Sorting (2/2)

- Sample run:

Input

```
Enter number of modules: 10
Enter module codes and student enrolment:
CS1010 292
CS1234 178
CS1012 358
CS2102 260
IS1103 215
IS2104 93
IS1112 100
GEK1511 83
IT2002 51
MAT1101 123
```

```
Sorted by student enrolment:
IT2002   51
GEK1511  83
IS2104   93
IS1112  100
MAT1101 123
CS1234  178
IS1103  215
CS2102  260
CS1010  292
CS1012  358
```

Output