Searching and sorting algorithms
M.A. El-dosuky

[1] What is the difference between searching and sorting algorithms?

A search algorithm is a method of locating a specific item in a larger collection of data.

A sorting algorithm is a technique for scanning an array and rearranging its contents in some specific order.

[2] Write the pseudo code for

a- Linear search

b- Binary search

Linear search

```
Set found to false.
Set position to -1.
Set index to 0.
While found is false and index < number of elements
If list [Index] is equal to search value
        found = true.
        position = index.
End If
Add 1 to index.
End While.
Return position
```

Binary search

```
    Set first Index to 0.
    Set last index to the last subscript In the array.
    Set found to false.
    Set position to -1.
    While found Is not true and first Is less than or equal to last
        Set middle to the subscript half-way between array[first] and array[last],
        If array [middle] equals the desired value
            Set found to true.
            Set position to middle.
        Else If array[middle] is greater than the desired value
            Set last to middle - 1.
        Else
            Set first to middle + 1.
        End If.
    End While.
    Return position.
```

[3] Write Algorithm for Recursive binary search identifying Precondition, Post condition

Preconditions: a is an array sorted in ascending order,
first is the index of the first element to search,
last is the index of the last element to search,
target is the item to search for.
If first > last
return failure
mid ← (first+last)/2
if a[mid] is equal to target
return mid
else if target < a[mid]
return result of recursive search of a from first up to mid-1
else
return result of recursive search of a from mid+1 up to last

Postcondition: value returned is position of target in a, otherwise return failure

[4] Write program for implementing Binary search for ascendingly sorted array, use:-

- first is the index of the first element to search,
- last. is the index of the last element to search,
- target is the item to search for.

```
int binarySearch(int a[ ], int first, int last, int target)
{
  if (first > last)
    return -1;       // -1 indicates failure of search
  int mid = (first+last)/2;
  if (a[mid] == target)
    return mid;
  else if (target < a[mid])
    return binarySearch(a, first, mid-1, target);
  else                  // target must be > a[mid]
    return binarySearch(a, mid+1, last, target);
 //Postcondition: Value returned is position of target in a otherwise -1 is returned
}
```

[5] Identify the errors in the following program and write the correct version;
The Code For Recursive Linear Search for array a with n elements to identify the target.

```
int linearSearch(int a[ ], int n, int target)
{
  // Precondition: a is an array indexed from 0 to n-1
  if (n <= 0)                    return -1;
  else {
       if (a[n-1] == target)     return n-1;
       else                      return linearSearch(a, n-1, target);
       }
}
```

[6] Trace the following

A- For an array a = {22, 11, 13, 81, 5}, trace linearSearch (a, 5,11).

a = {22, 11, 13, 81, 5}

- **linearSearch (a, 5,11)**
  - **n == 5; target == 11. Since a[4] != target, return linearSearch(a, 4, 11)**
- **linearSearch (a, 4,11)**
  - **n == 4; target == 11. Since a[3] != target, return linearSearch(a, 3, 11)**
- **linearSearch (a, 3, 11)**
  - **n == 3; target == 11. Since a[2] != target, return linearSearch(a, 2, 11)**
- **linearSearch(a, 2, 11)**
  - **n == 2; target == 11. Since a[l] == target, return 1**

B-For an array a = {22, 11, 13, 81, 5}, trace linearSearch (a, 5,50).

a = {22, 11, 13, 81, 5}

- **linearSearch (a, 5, 50)**
  - **n == 5; target == 50. Since a[4] != target, return linearSearch(a, 4, 50)**
- **linearSearch (a, 4,50)**
  - **n == 4; target == 50. Since a[3] != target, return linearSearch(a, 3, 50)**
- **linearSearch (a, 3, 50)**
  - **n == 3; target == 50. Since a[2] != target, return linearSearch(a, 2, 50)**
- **linearSearch (a, 2,50)**
  - **n == 2; target == 50. Since a[1] != target, return linearSearch(a, 1, 50)**
- **linearSearch (a, 1, 50)**
  - **n == 1; target == 50. Since a[0] != target, return linearSearch(a, 0, 50)**

1. **linearSearch (a, 0, 50)**
   - **n <= 0; return -1**

C- For an array a = {1, 6, 15, 22, 37, 45, 52}, trace binarySearch(a, 0, 6, 37).

- **binarySearch (a, 0, 6, 37)**
  first=0; last=6; target=37; mid=3; target > a[3], return binarySearch(a,4,6, 37)

- **binarySearch (a, 4, 6, 37)**
  first=4; last=6; target=37; mid=5; target < a[5], return binarySearch(a,4,4, 37)

- **binarySearch (a, 4, 4, 37)**
  first=4; last=4; target=37; mid=4; target == a[5], return 4

D- For an array a = {1, 6, 15, 22, 37, 45, 52}, trace binarySearch(a, 0, 6, 40).

- **binarySearch (a, 0, 6, 40)**

first=0; last=6; target=40; mid=3; target > a[3], return binarySearch(a,4,6, 40)

- **binarySearch (a, 4, 6, 40)**

first=4; last=6; target=40; mid=5; target < a[5], return binarySearch(a,4,4, 40)

- **binarySearch (a, 4, 4, 40)**

first=4; last=4; target=40; mid=4; target > a[4], return binarySearch(a,5,4, 40)

- **binarySearch (a, 5, 4, 40)**

first=4; last=4; target=40; first > last , return -1

[7] A- The function searchList shown below is an example of C++ code used to perform a linear search on an integer array. Identify the errors and rewrite the correct version

```
int searchList(int list[], int numElens, int value)
{
    int index =0;     //  a subscript to search array.
    int position = -1;   //  position of a search value.
    bool found = false;  // Flag to indicate if the value  was found.
    while (index < numElems && !found)
    {
    if (list[index] == value)   //  If the value is found
            {
                found = true;      // Set the flag
                position = index;      // Record the value's subscript
            }
            index++;   // Go to the next element
    }
    return position; // Return the position, or -1
 }
```

B- Write a program that uses the searchList function to search the five-element array tests [80, 90 , 60, 82, 100] to find a score of 100.

```
#include <iostream.h>
int searchList(int [ ], int, int);
const int arrSize = 5;

void main(void)
{
    int tests[arrSize] = (87, 75, 98, 100, 82};
    int results;
    results = searchList(tests, arrSize, 100);

    if (results == -1)
            cout « "You did not earn 100  on any test\n";
    else
    {
            cout « "You earned 100 points on test ";
            cout « (results + 1) « endl;
    }
 }
```

[8] what is the complexity of Linear and Binary Search?

Linear (Sequential, or serial) Search , if the size of array is n

- Best-Case: **O(1)** if item in [0]
- Worst-Case: **O(n)** if item in [n-1] or not found
- Average-Case **O(n/2)**

- Binary Search
  - average-case **O(log n)**
  - 2^n >= **no** of array element (**size**)
  - n is the no of comparisons if size = 8 then n=3

[9] True False

1. If data is sorted in ascending order, it means it is ordered from lowest to highest.**(true)**
2. If data is sorted in descending order, it means it is ordered from lowest to highest.**(false)**
3. The average number of comparisons performed by the linear search on an array of N elements is N/2 (assuming the search values are consistently found). **(true)**
4 . The maximum number of comparisons performed by the linear search on an array of N elements is N/2 (assuming the search values are consistently found). **(False→ N)**

[10] Write the pusedocode and program code (C++) for :-

1- Bubble Sort
2- Selection Sort

Bubble Sort

```
Do
    Set count variable to 0.
    For count is each subscript from 0 through the next-to-last subscript
        If array[count] is greater than array[count + 1]
            Swap  contents of array[count] and array [count + 1].
            Set swap flag to TRUE.
        End If.
    End For.
While any elements have been swapped.
```

```
void sortArray(int array[ ], int elems)
{
    int swap, temp;
    Do {
        swap = 0;
        for (int count = 0; count < (elems - 1); count++){
            if (array[count] > array[count + 1]) {
                temp = array[count];
                array [count] = array[count + 1];
                array [count + 1] = temp;
                swap = 1;
            }
        }
    } while (swap != 0);
}
```

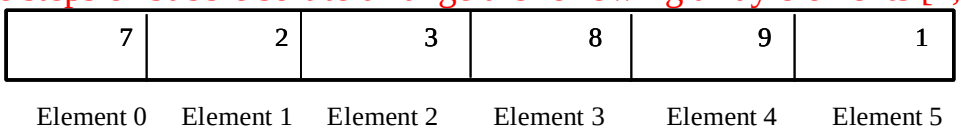
Selection Sort:

For startScan is set to each array subscript from 0 To next-to-last subscript

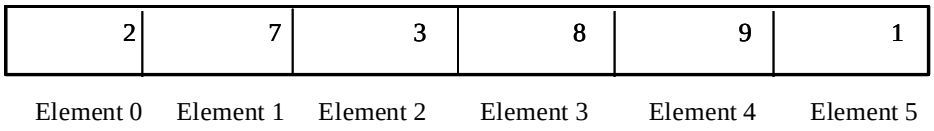
Set index variable to startScan.
Set minIndex variable to startScan.
Set minValue variable to array [startScan].
For index is each subscript In array from (startScan+1) to the next-to-last subscript
If array [index] is less than minValue
Set minValue to array[index].
Set minIndex to index.
End If.
Increment index.
End For.
Set array[minIndex] to array[startScan].
Set array [startScan] to minValue.

End For.

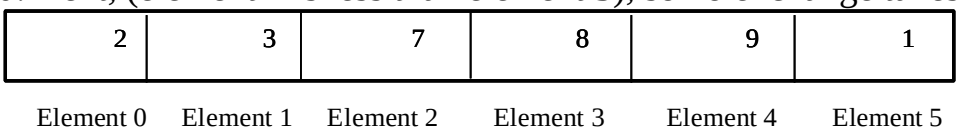
[11] Trace the steps of bubble sort to arrange the following array elements [7,2,3,8,9,1,**4**].



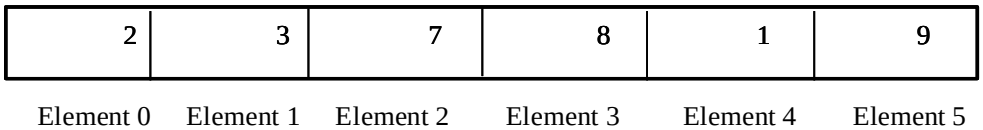
The bubble sort starts by comparing the first two elements in the array. If element 0 is greater than element 1, they are exchanged. After the exchange, the array shown above would appear as:



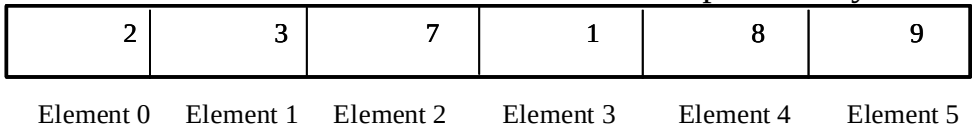
This method is repeated with elements 1 and 2. If element 1 is greater than element 2, they are exchanged. Next, (element 2 is less than element 3), so no exchange takes place.



Elements 3 and 4 are compared. no exchange. Elements 4 and 5 are compared, with exchange.

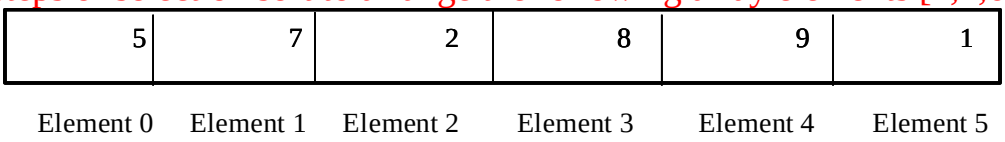


The entire array has been scanned, but its contents aren't in the right order yet. So, the sort starts again with elements 0 and 1, no exchange. Elements 1 and 2 are compared next, with no exchange. This continues until elements 3 and 4 are compared. they are exchanged.

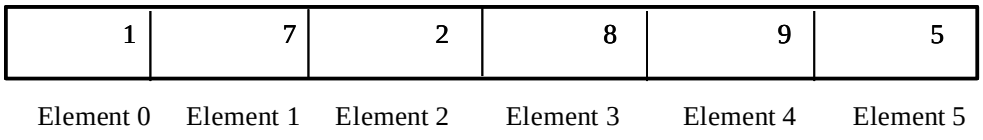


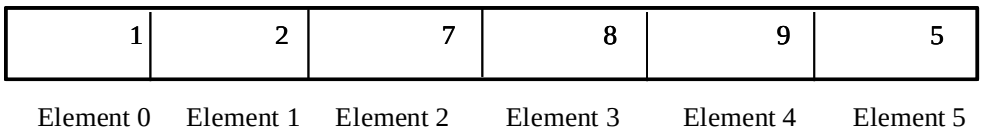
The sort repeatedly passes through the array until no exchanges are made

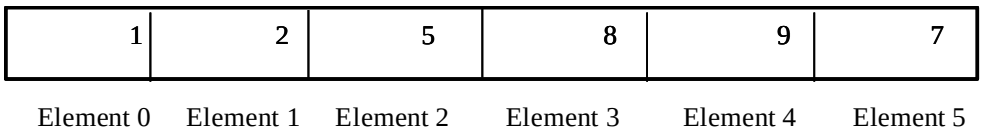
[12] Trace steps of selection sort to arrange the following array elements [7,2,6,8,9,1,4].

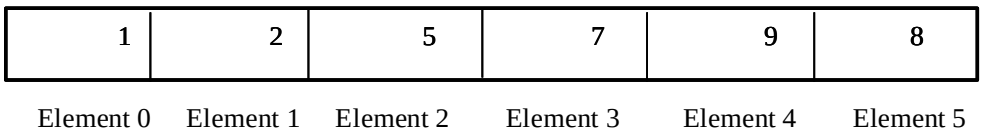


The selection sort scans the array, starting at element 0, and locates the element with smallest value (1 stored in element 5) . Its content then swapped with element 0.

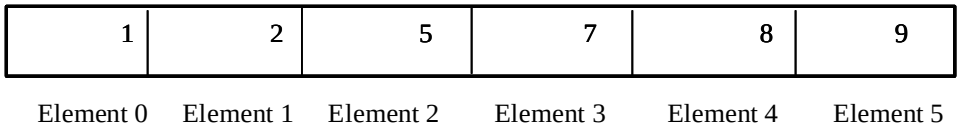








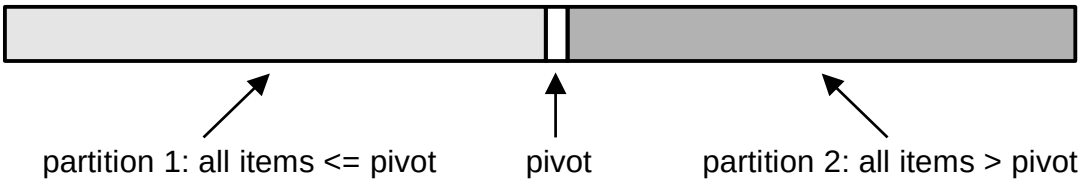
At this point there are only two elements left to sort. The algorithm finds that the value in element 5 is smaller than that of element 4, so the two are swapped. The array in its final arrangement:



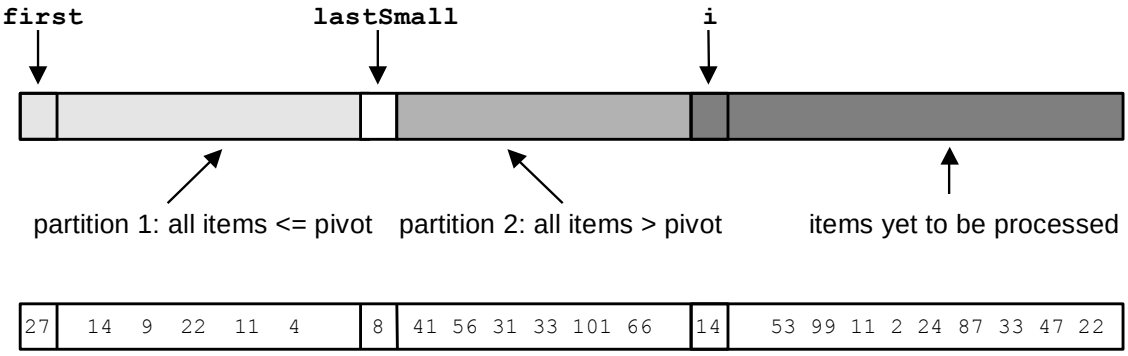
[13] A- Explain the basic idea of Quickisort **(Recursive Sorting Algorithm)**

Quicksort is a recursive sorting algorithm O(n log n) average-case performance.

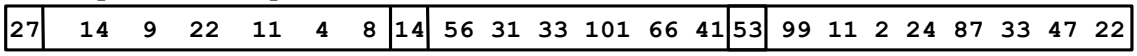
1. **pick one item** from the array to be sorted, and call it the pivot
2. **partition the items in the array around the pivot; that is recognize the array so that every item smaller than the pivot lies in the partition to the left of the pivot, and every item larger than the pivot lies in the partition to the right of the pivot**
3. **use recursion to sort the two partitions**



Partition:

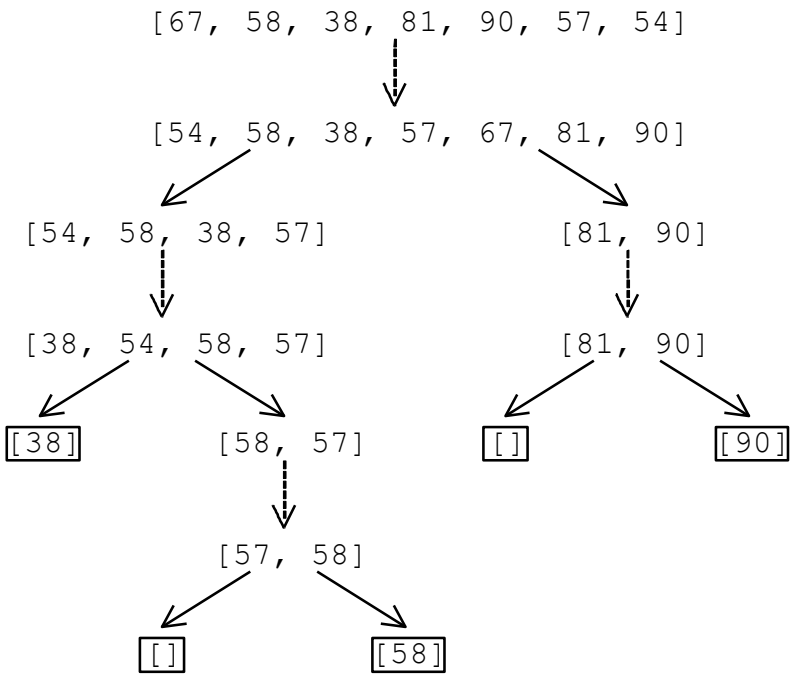


after swapElements update of lastSmall and i




```
void swapElements(int a[], int first, int last);
int partition(int a[], int first, int last)
{
  int lastSmall=first, i;
  for (i=first+1; i <= last; i++)
     if (a[i] <= a[first]) { // key comparison
       ++lastSmall;
       swapElements(a, lastSmall, i);
     }
  swapElements(a, first, lastSmall); // put pivot into correct position
  return lastSmall; // this is the final position of the pivot -- the split index
}
```

B- For the array a = {67, 58, 38, 81, 90, 57, 54}, trace quicksort(a,0, 6)



C-For the array a = {17, 22, 91, 11, 4, 50}, trace partition (a, 0, 5).

حلها بنفسك

D-For the array a = {11, 21, 19, 44, 56, 51, 95, 45}, trace partition (a, 4, 7).
حلها بنفسك

E- For the array a = {72,19, 25, 22, 19, 44, 91, 88, 11}, trace quicksort(a,0, 8)
حلها بنفسك

F- Draw a recursion tree to illustrate the operation of Quicksort on the following inputs:

1. A[] = [4,2,3,1,6,5,7]
2. A[] = [7,6,5,4,3,2,1]

حلها بنفسك

[14] Identify the errors in the following program and write the correct version;
The Code for the quicksort function

```
void quicksort(int a[ ], int first, int last)
{
  if (first >= last)
    return;
   // Otherwise, we're in the recursive case.
  // The partition function uses the item in a[first] as the pivot
  // and returns the position of the pivot -- split -- after the partition.
   int split = partition(a, first, last);
  // Recursively, sort the two partitions.
  quicksort(a, first, split-1);
  quicksort(a, split+1, last);
  // postcondition: a is sorted in ascending order between first and last
inclusive.
}
```

[15] MergeSort algorithm sorts an array with O(n log n) worst-case.
MergSort consists of

1. A recursive part that breaks the array into two subparts, makes recursive calls to sort each of the two subparts, and then calls a merge function to merge them back together:
2. A merge, that takes two sorted subarrays an merges them into a single sorted array.

```
MergeSort(a, first, last)
    Precondition:  Input is an array, a, and two indices first and last
    if (first >=last)
            then there's nothing left to be sorted, so just return
    else
            mid = (first + last)/2;
            MergeSort(a, first, mid)
            MergeSort(a, mid+1, last)
            Merge(a, first, mid, last)
    Postcondition:  Array a is sorted from index first to last
```

When writing Merge function:

```
            void merge(int a[], int first, int mid, int last)
```

[15] Towers of Hanoi puzzle consists of a set of graduated disks and tree pegs, as shown in FOLLOWING Figure. The goal is to move all the disks from peg 1 to peg 3, using peg 2 as necessary, and without ever putting a larger peg on top of a smaller peg.
First, move every disk except the largest from peg 1 to peg 2 (using the same recursive algorithm, but for a smaller problem). Then, move the largest disk to peg 3. Now, again using the recursive algorithm, move all the disks from peg 2 to peg 3.

When write the function:

void towers(int n, int fromPeg, int toPeg)

For example, towers(1, 1, 2) will print:

move disk from peg 1 to peg 2

and towers(3, 1, 3) will print:

move disk from peg 1 to peg 3
move disk from peg 1 to peg 2
move disk from peg 3 to peg 2
move disk from peg 1 to peg 3
move disk from peg 2 to peg 1
move disk from peg 2 to peg 3
move disk from peg 1 to peg 3