Sorting
$O(n^2)$ Algorithms

Dr. Tanvir Ahmed

Why we need sorting

- Sorting algorithms are fundamental problems in Computer Science
- We use sorting in our everyday life
- It makes searching easier
- It helps to analyze data easily.

$O(n^2)$ Algorithms

- We will learn number of sorting algorithms
  - Selection Sort
  - Insertion Sort
  - Bubble Sort
  - Merge Sort
  - Quick Sort.

- The first 3 algorithms are $O(n^2)$ Algorithms
- We assume that the input to the algorithm is an array of values (sorted or not)

Selection sort

- arr[] =

64	25	12	22	11
0	1	2	3	4

- **// Find the minimum element in arr[0...4]**
- // and place it at beginning
- **11** 25 12 22 **64**

- // **Find the minimum element in arr[1...4]**
- // and place it at beginning of arr[1...4]
- 11 **12 25** 22 64

- **// Find the minimum element in arr[2...4]**
- // and place it at beginning of arr[2...4]
- 11 12 **22 25** 64

- // Find the minimum element in arr[**3**...4]
- // and place it at beginning of arr[3...4]
- 11 12 22 25 64

Now write a function to solve it!

```
void selectionSort(int arr[], int n)
{



}
```

```
void selectionSort(int arr[], int n)
{
    int i, j, min_idx, temp;

    // One by one move boundary of unsorted subarray
    for (i = 0; i < n-1; i++)
    {
        // Find the minimum element in unsorted array
        min_idx = i;
        for (j = i+1; j < n; j++)
          if (arr[j] < arr[min_idx])
            min_idx = j;

        // Swap the found minimum element with the first element
       temp = arr[i];
        arr[i] = arr[min_idx];
        arr[min_idx] = temp;

     }
}
```

Selection Sort:

- **Analysis of Running Time:**
- During the first iteration
  - We “go through” all n items searching for the minimum
  - This is essentially n simple steps

- During the second iteration, *i* starts at index 1
  - We “go through” n – 1 items searching for the minimum
  - We do not need to account for the item at index 0 (it is already in correct position)
- During the third iteration:
  - We “go through” n – 2 items searching for the minimum
  - We do not need to account for the items at index 0 and 1 (they are already sorted)

Selection Sort:

- Analysis of Running Time:
- 4^{th} iteration:
  - We will “go through” n – 3 steps
- 5^{th} iteration
  - We will “go through” n – 4 steps
- …
- Final iteration
  - There will simply be one step
- We can add up the TOTAL number of simple steps
- TOTAL = n + (n-1) + (n-2) + (n-3) + … + 3 + 2 + 1
- Can you guess how many steps in total?
- Maybe writing in revere order will help you to guess.
- We learned it while doing summation
- You are addin 1 to n and it is
  - n(n+1)/2

Bubble sort

- **Bubble Sort:** sometimes referred to as **sinking sort**,
- Basic idea: the heavy items are going down gradually
- You always compare consecutive elements
    - Going left to right
- Whenever two elements are out of place,
    - SWAP them so that the larger item is moving next
- At the end of a single iteration,
    - the maximum element will be in the last spot
- Now you simply repeat this n times
    - where n is the number of elements being sorted
- On each pass, one more maximal element will be put into place

Bubble Sort:

- Example:
  - Here is an array of 8 integers: 6, 2, 5, 7, 3, 8, 4, 1
  - On a single pass of the algorithm, here is the state of the array:

2, 6, 5, 7, 3, 8, 4, 1
2, 5, 6, 7, 3, 8, 4, 1
2, 5, 6, 7, 3, 8, 4, 1
2, 5, 6, 3, 7, 8, 4, 1
2, 5, 6, 3, 7, 8, 4, 1
2, 5, 6, 3, 7, 4, 8, 1
2, 5, 6, 3, 7, 4, 1, 8 (8 is now in place!)

The "swapped" elements are underlined.

Of course, a swap only occurs as needed.

Note:
We'd have to do this **EIGHT** more times to guarantee a sorted list!

Bubble sort full steps examples

- **First Pass:**
  (**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
  (1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4
  (1 4 5 2 8) –> (1 4 2 5 8), Swap since 5 > 2
  (1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them. *//8 is in place after this iteration and we don't need to check the last index anymore*
- **Second Pass:**
  (**1 4** 2 5 8) –> (**1 4** 2 5 8)
  (1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2
  (1 2 **4 5** 8) –> (1 2 **4 5** 8) //5 is also *is in place after this iteration and we don't need to check the last **TWO** index anymore*
- Now, the array is already sorted, but the our algorithm does not know and it will keep iterating to find largest number in that iteration. However, no swapping will happen as the data are already sorted.
- **Third Pass:**
  (**1 2** 4 5 8) –> (**1 2** 4 5 8)
  (1 **2 4** 5 8) –> (1 **2 4** 5 8) //4 is in place, no need to check last THREE items anymore in next pass
- **Fourth Pass:**
  (**1 2** 4 5 8) –> (**1 2** 4 5 8)

//2 is in place, no need to check last 4 items anymore

So, as our array has 5 elements and 4 places are fixed,

no need further iteration

Now write a function to solve it!

```
void bubbleSort(int arr[], int n)
{


}
```

Example courtesy: https://www.geeksforgeeks.org/

Bubble sort implementation in C

```
void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
   int i, j;
   for (i = 0; i < n-1; i++)

       // Last i elements are already in place
       for (j = 0; j < n-i-1; j++)
           if (arr[j] > arr[j+1])
              swap(&arr[j], &arr[j+1]);
}
```

```
/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", arr[i]);
    printf("n");
}

// Driver program to test above functions
int main()
{
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]); // you can extract the array size by this line
    bubbleSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

Complexity analysis of Bubble sort

- *When i = 0 it iterates n-1 times*
- *When i = 1 it iterates n-2 times*
- *When i = 2 it iterates n-3 times*
- Final iteration: There will simply be one step

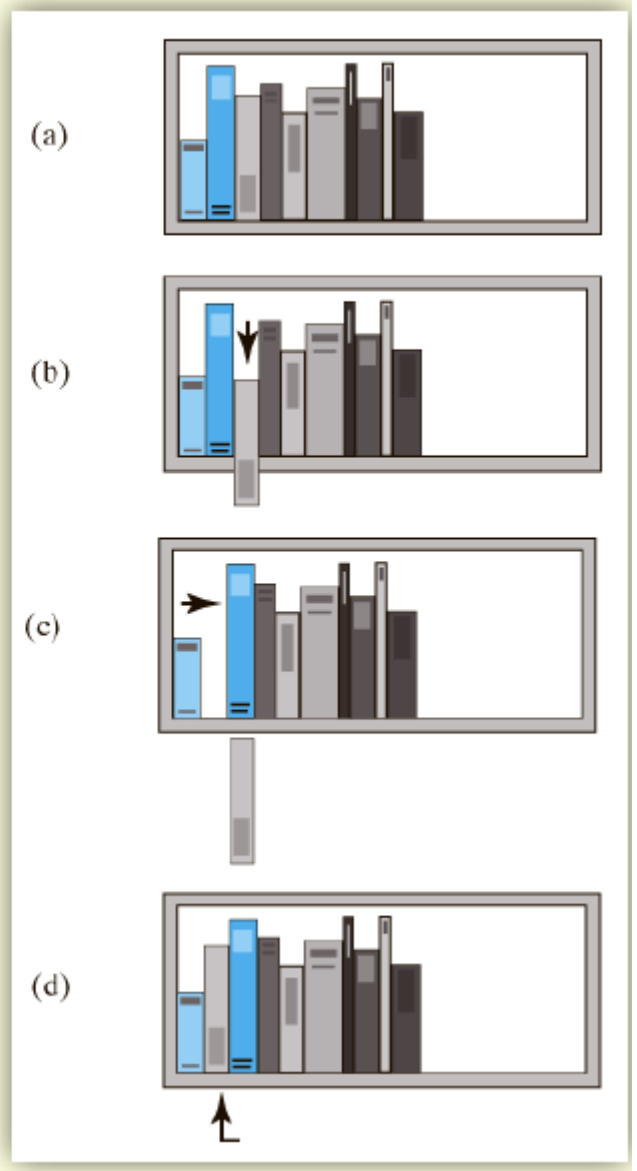
- Total = (n-1) + (n-2) + (n-3) + … + 3 + 2 + 1
- So, adding natural numbers for 1 to n-1
- We know 1+2+3+....+n = n(n+1)/2
- Now in this case it is up to n-1, so *total = (n-1)(n-1+1)/2 =n(n-1)/2*
- Order $O(n^2)$

Insertion sort

- Human generally apply this sorting technique when sorting documents.
- Think about sorting books of different sizes placed in a book shelf
- **Bookshelf example:**
- If first two books are out of order:
  - Remove second book
  - Slide first book to right
  - Insert removed book into first slot
- Next, look at third book, if it is out of order:
  - Remove that book
  - Slide 2nd book to right
  - Insert removed book into 2nd slot
- Recheck first two books again
- Etc.

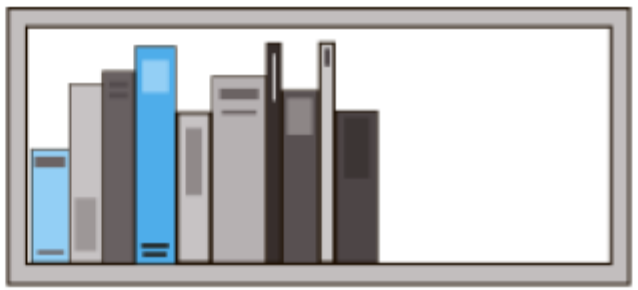
Insertion Sort:

- Bookshelf example:
  - This picture shows the “insertion” of the third book
    - The 3rd book is removed
    - It is compared with the 2nd book
    - The 2nd book is larger
    - So we slide the 2nd book into the 3rd spot
    - We then compare our original 3rd book with the 1st book
    - They are in order
    - So we simply insert the original 3rd book in the 2nd spot



Insertion Sort:

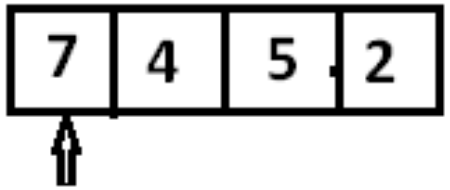
- Bookshelf example:
  - In general:



1. Remove the next unsorted book.
2. Slide the sorted books to the right one by one until you find the right spot for the removed book.
3. Insert the book into its new position.

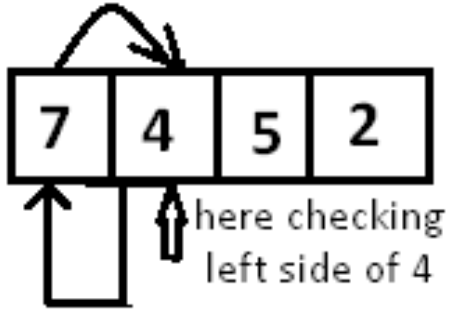
Insertion Sort steps

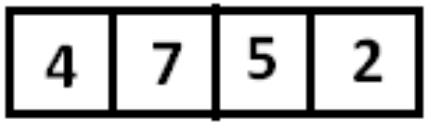
STEP 1.



No element on left side of 7,so no change in its position .

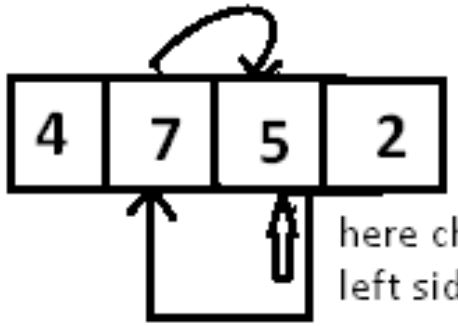
STEP 2.

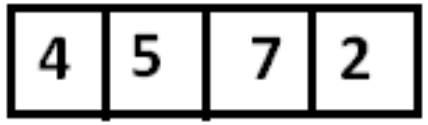




As 7>4 ,therfore 7 will be moved forward and 4 will be moved to 7's position.

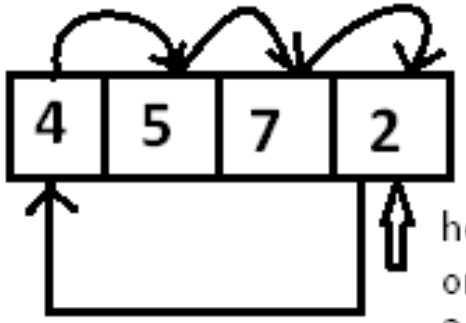
STEP 3.

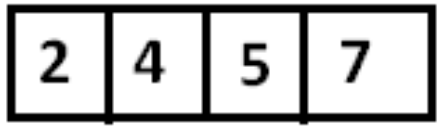




As 7>5,7 will be moved forward,but 4 <5,so no change in position of 4. And 5 will be moved to position of 7.

STEP 4.





As all the element on left side of 2 are greater than 2,so all the elements will be moved forward and 2 will be shifted to position of 4

Image source: https://www.hackerearth.com/practice/algorithms/sorting/insertion-sort/tutorial/

More example of insertion sort

- **12**, 11, 13, 5, 6
- Let us loop for i = 1 (second element of the array) to 4 (last element of the array)
- i = 1. Since 11 is smaller than 12, move 12 and insert 11 before 12
  11, 12, 13, 5, 6
- i = 2. 13 will remain at its position as all elements in A[0..i-1] are smaller than 13
  11, 12, 13, 5, 6
- i = 3. 5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.
  5, 11, 12, 13, 6
- i = 4. 6 will move to position after 5, and elements from 11 to 13 will move one position ahead of their current position.
  5, 6, 11, 12, 13
- **Now try to write the code! (But you don't have to submit it)**

Insertion sort implementation in C

```
void insertionSort(int arr[], int n)
{
    int i, item, j;
    for (i = 1; i < n; i++)
    {
        item = arr[i];

       /* Move elements of arr[0..i-1], that are
        greater than key, to one position ahead
        of their current position */
        for(j=i-1; j>=0; j--)
        {
            if(arr[j]>item)
                arr[j+1] = arr[j];
            else
                break;
         }
        arr[j+1] = item;
    }
}
```

```
/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", arr[i]);
    printf("n");
}

// Driver program to test above functions
int main()
{
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]); // you can extract the array size by this line
    bubbleSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

Insertion sort complexity analysis

- Considering only number of times the loop will run in worst case:
- i starts from 1 and ends at n-1
- When:
- i = 1 will take 1 iteration
- i = 2 will take 2 iteration
- i = 3 will take 3 iteration
- i = 4 will take 4 iteration
- i = n-1 will take n-1 iteration

- 1 + 2 + 3 + 4 + …..+ n-1
- We know 1+2+3+....+n = n(n+1)/2
- Now in this case it is up to n-1, so *total = (n-1)(n-1+1)/2 =n(n-1)/2*
- Order $O(n^2)$
- So it is quadratic time.

-In our next lab we will experimentally evaluate the run time of various sorting algorithms.
-We will also plot the runtime to visualize the result

Acknowledgement and more notes

- Some slides are taken from lecture note of Dr. Jonathan Cazalas: https://www.cs.ucf.edu/courses/cop3502/spr2012/notes/COP3502_26_N-SquaredSorts.pdf

- **More Notes:**
- Prof Arup's notes on Sorting (read on your own): http://www.cs.ucf.edu/~dmarino/ucf/transparency/cop3502/lec/NsqrdSorts-19.doc