Lecture 3: Elementary Sorting Algorithms

29th August, 2023

Selection Sort

⇒ First, find the smallest item in the array and exchange it with the first entry (itself if the first entry is already the smallest). Then, find the next smallest item and exchange it with the second entry. Continue in this way until the entire array is sorted.

⇒ This method is called selection sort because it works by repeatedly selecting the smallest remaining item.

Pseudo Code

```
function sort(array){
    for i in 0...N {
        min_ele = find_min(array, i, N)
        exchange min_ele with i’th index in array
    }
}
```

Implementation

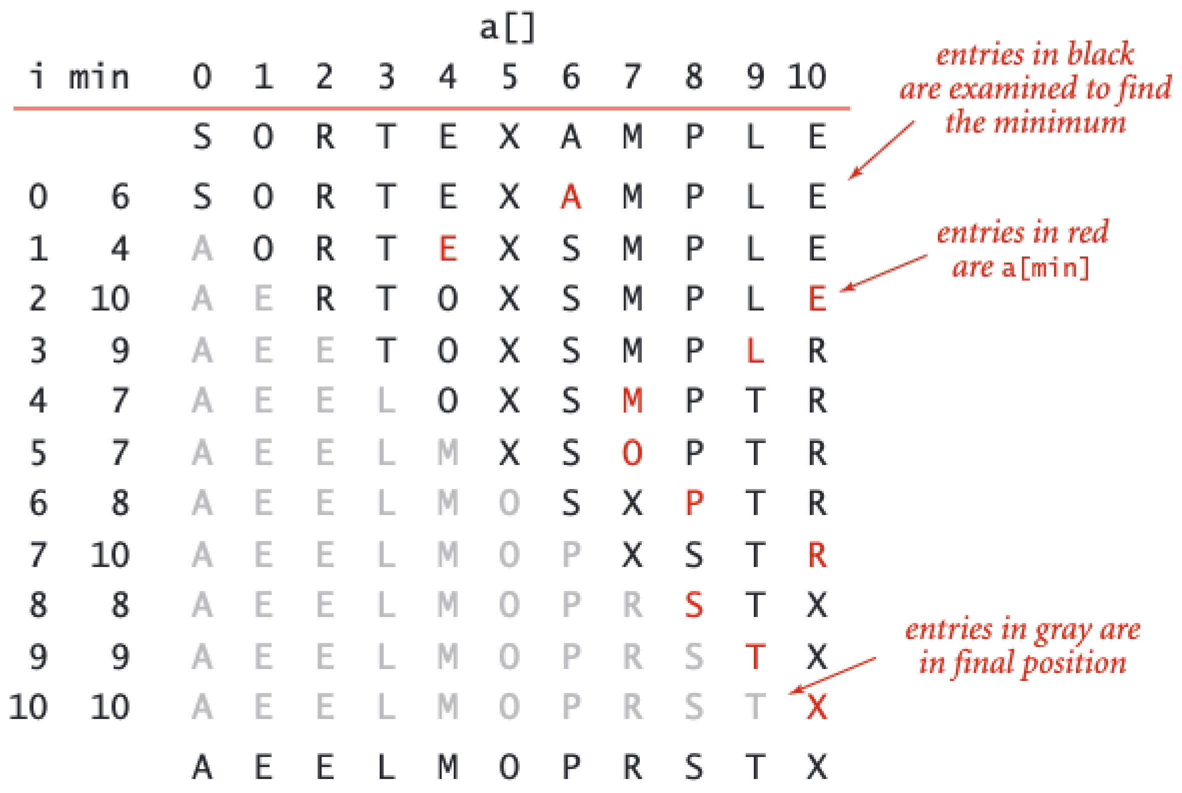
```
void selection_sort(vector<int> nums){
     int n = nums.size();
     for(int i=0; i<n; i++){

         // finding minimum element
         int minIndex = i;
```

```
        for(int j=i+1; j<n; j++){
            if(nums[j] < nums[minIndex]) minIndex = j;
        }

        // put minimum number at it's actual index
        swap(nums[i], nums[minIndex])
    }
}
```

Example & Visualization



		a[]										
i	min	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
0	6	S	O	R	T	E	X	A	M	P	L	E
1	4	A	O	R	T	E	X	S	M	P	L	E
2	10	A	E	R	T	O	X	S	M	P	L	E
3	9	A	E	E	T	O	X	S	M	P	L	R
4	7	A	E	E	L	O	X	S	M	P	T	R
5	7	A	E	E	L	M	X	S	O	P	T	R
6	8	A	E	E	L	M	O	S	X	P	T	R
7	10	A	E	E	L	M	O	P	X	S	T	R
8	8	A	E	E	L	M	O	P	R	S	T	X
9	9	A	E	E	L	M	O	P	R	S	T	X
10	10	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

entries in black are examined to find the minimum
entries in red are a[min]
entries in gray are in final position

Trace of selection sort (array contents just after each exchange)

Complexity Analysis

Time Complexity

The process of finding the minimum element from an array or exchanging elements is not related to input. For any case, time taken by those The algorithm is **not dependent** on input.
So, we can say,

Best Case = Worst Case = Average Case

Now, let's calculate the time complexity.
In every iteration we find the minimum element and replace it with its correct
position.

Let n be the size of the array,
Finding minimum requires a whole array traversal. → n steps
Swapping the minimum number to its correct position in array is → 1 step

Now, on every iteration we remove one element from array

$$\begin{aligned} \text{total steps} &= (n + 1) + (n-1 + 1) + (n-2 + 1) + \dots + (1 + 1) \\ &= (n + n-1 + n-2 + \dots + 1) + (1 + 1 + 1 + \dots + 1) \\ &= (n*n-1)/2 + n \\ &= (n^2 - n)/2 + n \\ &= \theta(n^2) \end{aligned}$$

⇒ Time Complexity: $\theta(n^2)$

Space Complexity

The algorithm is not creating any other variables to sort the array.

⇒ Input Space: $\theta(n)$
⇒ Output Space: $\theta(1)$
⇒ Auxiliary Space: $\theta(1)$

Insertion Sort

⇒ To sort an array of size N in ascending order, iterate over the array and compare the current element to its predecessor(elements before current one),
if the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

⇒ As in selection sort, the items to the left of the current index are in sorted order during the sort, but they are not in their final position, as they may have to be moved to make room for smaller items encountered later. The array is, however, fully sorted when the index reaches the right end.

⇒ Values from the unsorted part are picked and **inserted** at the correct position in the sorted part.

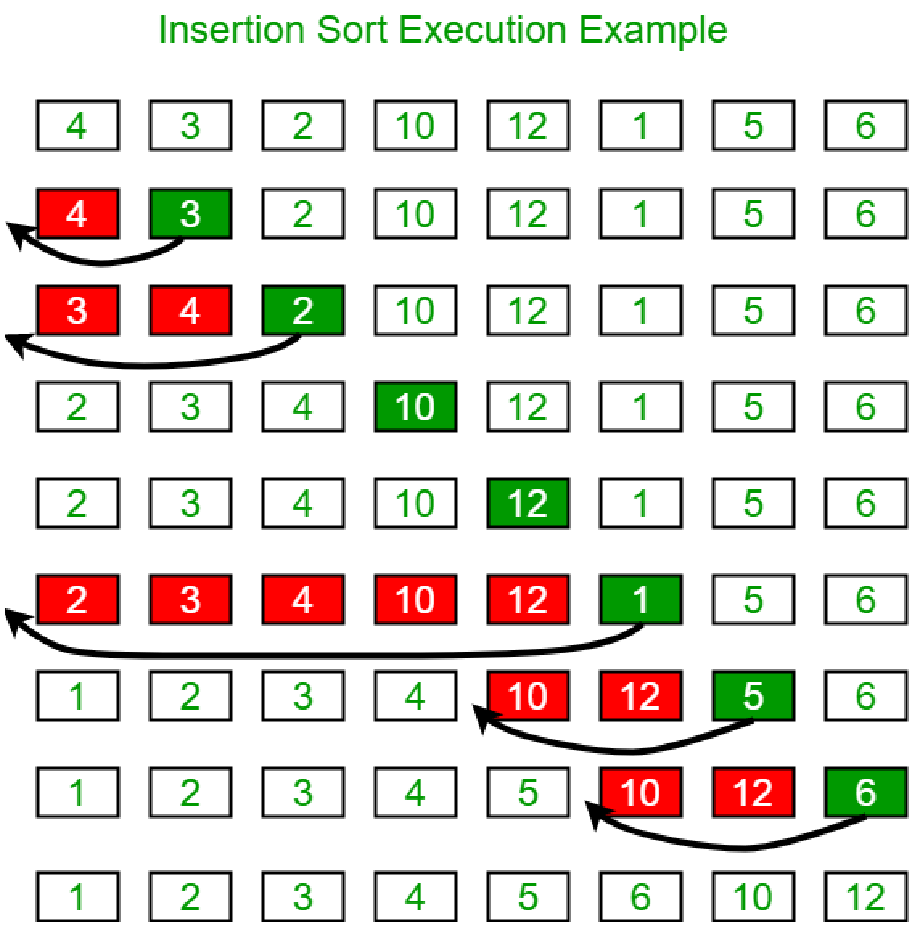
Pseudo Code

```
function insertion_sort(array){
    for i in 0...N {
        pos = find_correct_poision(array, 0, i-1)
        insert i'th element at pos in array
    }
}
```

Implementation

```
void insertion_sort(vector<int> nums){
      for (int i = 0; i < n; i++) {
            current = nums[i];
            int prev_ind = i - 1;
           // inserting current element to its correct position
           while (prev_ind >= 0 && arr[prev_ind] > current) {
               // shifting
               arr[prev_ind + 1] = arr[prev_ind];
               prev_ind -= 1;
           }
           // writing the element
           arr[prev_ind + 1] = current;
      }
}
```

Example & Visualization



Complexity Analysis

Time Complexity

Every iteration, the insertion sort finds the correct position of i'th element in the currently sorted array. To do so, it starts by looking at the last element and goes back until the current element is bigger. Now, let's do case study.

- **Case 1**: nums = [1,3,4,6,8] element to insert = 5
  It will take 3 steps to find the correct position of element 5.
- **Case 2**: nums = [1,3,4,6,8] element to insert = 0
  It will take 6 steps to find the correct position of element 0.
- **Case 3**: nums = [1,3,4,6,8] element to insert = 10
  It will take 1 step to find the correct position of element 10.

So, we can say that the number of steps to find the correct position depends on the input.

Best Case

When the element to be inserted is the largest among all currently sorted elements. So, just in the first step insertion sort algorithm knows that the element will be placed at the end.
e.g. nums = [1,3,4,6,8] element to insert = 10
⇒ Best Case Time Complexity of **inserting element**: $\theta(1)$

Now, if the best case happens in every iteration, the algorithm will take
$\theta(1) + \theta(1) + \ldots + \theta(1) = \theta(n)$
As the outer loop runs for n times.

Best case will happen when the array insertion sort tries to sort is already sorted.
e.g. nums = [1,3,4,6,8]
⇒ Best Case Time Complexity of **Insertion Sort**: $\theta(n)$

Worst Case

When the element to be inserted is the minimum among all current sorted elements. So, the insertion sort algorithm will iterate through all the numbers to finalize the current element's position.
e.g. nums = [1,3,4,6,8] element to insert = 0

⇒ Worst Case Time Complexity if **inserting element**: θ(n)

Now, if the worst case happens in every iteration, the algorithm will take
θ(n) + θ(n) + … + θ(n) = θ(n^2)

Worst case will happen when the array insertion sort tries to sort is sorted in reverse order.
e.g. nums = [8,6,4,3,1]
⇒ Worst Case Time Complexity of **Insertion Sort**: θ(n^2)

Space Complexity

The algorithm is not creating any other variables to sort the array.

⇒ Input Space: θ(n)
⇒ Output Space: θ(1)
⇒ Auxiliary Space: θ(1)

Selection VS Insertion Sort

⇒ Worst Case Time Complexity of both algorithms are the same, does that mean we can pick any algorithm and it will behave the same?
No, even though the worst-case is the same, best-case time complexity is different.
If you pass partially sorted array as the input to both algorithms,

- Insertion Sort will take linear time
- Selection Sort will still take quadratic time

The analysis of algorithms does not stop on just time complexity or space complexity.
Let’s compare the number of exchanges of elements and comparisons.

Number of Exchanges

Selection Sort

```
int n = nums.size();
for(int i=0; i<n; i++){

     int minIndex = i;
     for(int j=i+1; j<n; j++){
         if(nums[j] < nums[minIndex])
         minIndex = j;
     }

    swap(nums[i], nums[minIndex])
}
```

Insertion Sort

```
for (int i = 0; i < n; i++) {
  current = nums[i];
  int prev_ind = i - 1;

  while (prev_ind >= 0 &&
         arr[prev_ind] > current)
  {
    arr[prev_ind + 1] = arr[prev_ind];
    prev_ind -= 1;
  }

  arr[prev_ind + 1] = current;
}
```

The highlighted part is exchanging the array elements.
⇒ In selection sort, the exchange only happens **once** in an iteration. So, we can say, the selection sort will do $\theta(n)$ exchanges.
⇒ In insertion sort, the exchange happens in a loop. Considering the worst-case scenario, the loop might run for n steps. so, we can say in the worst-case the insertion sort may exchange up to $\theta(n^2)$ exchanges. But in the best-case scenario, the loop will take constant time, so the total exchanges in best-case will be $\theta(n)$.

Number of Comparisons

Selection Sort

```
int n = nums.size();
for(int i=0; i<n; i++){

     int minIndex = i;
     for(int j=i+1; j<n; j++){
         if(nums[j] < nums[minIndex])
```

Insertion Sort

```
for (int i = 0; i < n; i++) {
  current = nums[i];
  int prev_ind = i - 1;

  while (prev_ind >= 0 &&
         arr[prev_ind] > current)
```

```
        minIndex = j;
      }

    swap(nums[i], nums[minIndex])
}
```

```
  {
    arr[prev_ind + 1] = arr[prev_ind];
    prev_ind -= 1;
  }

  arr[prev_ind + 1] = current;
}
```

The highlighted part is where comparisons are happening in the algorithm.
⇒ In the selection sort, in every iteration, the number of comparisons are directly proportional to how many steps the loop will take. As we have already comparisons will be $\theta(n^2)$
seen previously, the loop will take θ(n-i) steps. So adding them up, total ⇒ In the insertion sort, in every iteration, the number of comparisons are directly proportional to how many steps the loop will take. As we have already seen previously, the number of steps the loop will run depends on the input. Considering the worst-case scenario, the number of comparisons will be $\theta(n^2)$. But in the best-case scenario, the number of comparisons will be $\theta(n)$.

Real Life Examples

	Number of Exchanges		**Number of Comparisons**	
	Selection Sort	Insertion Sort	Selection Sort	Insertion Sort
Worst Case	$\theta(n)$	$\theta(n^2)$	$\theta(n^2)$	$\theta(n^2)$
Best Case	$\theta(n)$	$\theta(n)$	$\theta(n^2)$	$\theta(n)$
Better	Selection Sort		Insertion Sort	

⇒ Why do we care about the number of exchanges and comparisons?
In some cases, these operations are not as easy as arrays.
Let's take some real-life examples,

- If we are running Insertion Sort to sort the boxes by size for amazon. The number of exchanges might go up to $\theta(n^2)$. Which will take more labor work. On the other hand if we had used Selection Sort, the exchanges would be far less compared to Insertion Sort.
- If we are running Selection Sort on the array of strings, the number of comparisons will be $\theta(n^2)$. But, the one string comparison is not similar to one integer comparison. To compare the string, we need θ(string len) steps. So, instead if we would have used Insertion Sort we might have saved some time.

Resources

- Book: Algorithms, Fourth Edition By Robert Sedgewick and Kevin Wayne.
  Section 2.1
- Algorithm Visualization
- Polls