Lecture 14

Sorting
Insertion Sort
Selection Sort

October 07, 2021
Thursday

SORTING

- Efficiency can be increased if the data is sorted according to some criteria.
- Convenience of using sorted data is unquestionable.
  - Just think of finding a word in the dictionary when words are not sorted alphabetically.
- In some scenarios it is essential to sort data before further processing.
- Select a criteria that will be used to order the data.
  - Application dependent.
  - Ascending / Descending / Alphabetically / Alpha Numeric

SORTING

- Once a criterion is selected
  - How to put a set of data in order using that criterion.
- How we will decide which method is best?
  - Certain criteria for efficiency have to be established.
  - Method for quantitatively comparing different algorithms must be chosen.
  - Comparison should be machine independent.
- Each sorting algorithm have following two properties
  - Number of comparisons
  - Number of data movements.
- The efficiency of these two operations depends on the size of the dataset.

SORTING

- Determining the exact number of comparisons is not always necessary or possible.
    - An approximation is preferred.
    - The number of comparisons and movements is approximated with big-O notation
- In addition we are interested in knowing how intelligent the algorithm is
    - E.g., How much time does the machine spend on data ordering if the data are already sorted.
    - Does it recognize the initial ordering immediately
    - Or it is completely unaware of the state of the data.

SORTING

- Therefore, we compute the number of comparisons and movements (if possible)
  - Best Case (Data already sorted)
  - Average Case (Data in random order)
  - Worst Case (Data in reverse order)

- The number of comparison and number of movements do not have to coincide.
  - An algorithm may perform very well with comparisons
  - But performs poorly with movements or vice versa.

- Comparisons are inexpensive if we are comparing simple integers/characters
  - But what if we are comparing strings, or arrays of numbers.
  - Then we are more concerned about the number of comparisons.

SORTING

- Similarly, if data items to be moved are large, such as structures & objects
  - We will be concerned about the number of movements required.
- If sorting is used rarely for small set of data
  - Then using a simpler algorithm is more suitable than a more complex and a little more efficient algorithm.
  - If sorting is done frequently on larger dataset then an efficient algorithm despite being more complex will be more beneficial.

INSERTION SORT

INSERTION SORT

- Sorting starts by considering the two first elements of the array named **data**.
- If they are out of order an interchange takes place.
- Then third data item is considered
  - If it is smaller than both items data[0] and data[1].
    - Then both data[0] and data[1] are shifted.
  - If it is smaller than data[1] but greater than data[0].
    - Then only data[1] is shifted.
  - If it is greater than both items data[0] and data[1]
    - No movements take place.

INSERTION SORT

```
void InsertionSort(int data[ ], int n){
    for(int i = 1, j; i<n; i++){
        int tmp =  data[i];
        for(j = i; j > 0 && tmp < data[ j - 1]; j--)
            data[j] = data [ j-1];
        data[j] = tmp;
    }

}
```

INSERTION SORT

- Sorts the array only when it is really necessary.
  - If the array is in order, no substantial movements are performed.
  - variable tmp is initialized and its value is moved back to the same position.
  - The algorithm recognizes that part of array is already sorted and stops execution accordingly.
- The fact that elements may be in their proper position is overlooked.
  - They can be moved from their positions and later moved back.

- If an item is being inserted all the items greater than that element have to be moved.

INSERTION SORT

- The outer loop always performs (n -1) iterations
  - however, the number of elements greater than data[i] to be moved are not always the same.

- The best case is when the data is already sorted.
  - Only one comparison is made for each position i, which is $\boldsymbol{O}$ (n).
  - 2 (n - 1) moves are performed, all of them redundant.

- The worst case is when data is in reverse order.

  - For each i data[i] is less than every data[0], … , data[i-1], hence, each of them is moved by one position.
  - For each iteration of i of the outer loop there are i comparisons.
  - Total comparisons n (n - 1) / 2 = $\boldsymbol{O}(n^2)$
  - number of assignments is also $\boldsymbol{O}(n^2)$
  - The average case is also $\boldsymbol{O}(n^2)$

SELECTION SORT

SELECTION SORT

- Selection sort localize the exchanges of array elements.
  - Finds the misplaced element and put it in its final position.
  - The element with the lowest value is selected and exchanged with the element in the first position.
  - Then the next smallest element is found and exchanged with second position.

SELECTION SORT

```
void SelectionSort(int data[ ], int n){
    for(int i = 0, j, least; i < n - 1; i++){
        for( j = i + 1, least = i; j < n; j++)
            if (data [ j ] < data [least]
                least = j;
        swap (data [ least ], data [ i ] );
    }

}
```

SELECTION SORT

- Swap exchanges data [least] and data [i].
- Least is not the smallest element but the position of smallest element.

- The outer loop executes for n - 1 times.
  - For each i between 0 and n - 2
    - inner loop iterates (n - 1) - i times.

- Since comparisons are done in inner loop
  - There are $n(n - 1) / 2 = \boldsymbol{O}(n^2)$
  - This number stays same for all cases.

- We can save some swaps by checking if i and least are same or not?