Lecture 15
Bubble Sort

October 08, 2021
Friday

BUBBLE SORT

BUBBLE SORT

- Best understood if we visualize array as a vertical column.
  - Smallest element are at the top.
  - Largest elements are at the bottom.
- The array is scanned from bottom to top.
  - Two adjacent elements are interchanged if they are found out of order with respect to each other.
  - First data[n - 1] and data[n - 2] are compared and swapped if required.
  - Next data[n - 2] and data[n - 3] are compared and swapped if required.
  - Upto data[1] and data[0] are compared and swapped if required.
- This way the smallest element is bubbled up to the top.

INSERTION SORT

- This is the end of pass 1 and one element is sorted.
- In the second pass, the array is scanned again.
  - Comparing consecutive items and interchanging them if required.
  - However, this time the last comparison is done for data[2] and data[1].
  - Because the smallest element is already in its proper position.
  - The second pass bubbles the second element to its position.
- The process continues until the last pass
  - Only one comparison data[n - 1] and data[n - 2] and possibly one interchange is performed.

BUBBLE SORT

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

BUBBLE SORT

- The number of comparisons are same in each case
  - Best Case Comparisons: $n (n - 1) / 2 = \boldsymbol{O} (n^2)$
  - Average Case Comparisons: $n (n - 1) / 2 = \boldsymbol{O} (n^2)$
  - Worst Case Comparisons: $n (n - 1) / 2 = \boldsymbol{O} (n^2)$

- The number of moves in Average case:
  - If an *i-cell* array is in the random order, then the number of swaps can be any number between 0 and $i - 1$.
  - Either no swap, or $i - 1$ swaps.
  - The average number of swaps for an iteration is: $(n - i - 1) / 2$
  - Adding all the subarrays passes total swaps become: $3 / 4\ n (n - 1)$

BUBBLE SORT

- The main disadvantage is that it painstakingly bubbles items step by step towards the top of the array.
- If an element has to be moved from bottom to top it will be exchanged with every element in the array.
  - It does not skip the elements like selection sort.
  - Some items will be moved back and forth, even though they were in their correct position from the beginning.

BUBBLE SORT

- In average case bubble sort makes approximately twice as many comparisons and almost twice number of movements as insertion sort.
- As many comparisons as selection sort and n times more moves than selection sort.
- It could be said that insertion sort is twice as fast as bubble sort.

BUBBLE SORT IMPROVED

```
void BubbleSort2(int data[ ], int n){
    bool again = true;
    for(int i = 0; i < n-1 && again; i++) {
        for(int j = n - 1, again = false; j > i; --j)
            if ( data [ j ] < data [ j - 1]) {
                swap ( data[ j ], data [ j - 1] );
                again = true;
            }
    }
}
```

BUBBLE SORT2

- The flag is in outer loop, and needs to be true for outer loop to continue.
- If no swap in the complete iteration of inner loop occurs
  - This indicates that all elements are in their appropriate position
  - The outer loop terminates without running for n-1 items.
  - This may saves significant amount of iterations in some scenarios.
  - This is most likely to happen, when items are relatively closer to their final position and array is sorted in first few iterations.
  - As array gets sorted the iterations are stopped

BUBBLE SORT2

- The improvement is insignificant in worst case scenario and behaves just like the original one.
- The worst case for the number of comparisons is when
  - The largest number is at first position.
  - As it will move down one step in each pass.
  - Very seldom the flag is useful.
- Since, an additional variable needs to be maintained by BubbleSort2 it becomes slower than the original BubbleSort.
- But the idea is further refined for Comb Sort which is faster than Bubble Sort.

VISUALIZATION


a[0]
a[1]
a[2]
a[3]
a[4]
a[5]
a[6]
a[7]
a[8]
a[9]
0
1
3
7
2
5
4
8
6
9