CSC 401

Lectures on
Analysis of Algorithms

by

Dr. Marek A. Suchenek ©

Computer Science
CSUDH

CSC 401

Lecture 8
Sorting

Running Times, Lower Bounds, Optimality

Sorting

Insertion Sort

Insertion Sort

Figure 4.1 Unsorted elements

Insertion Sort

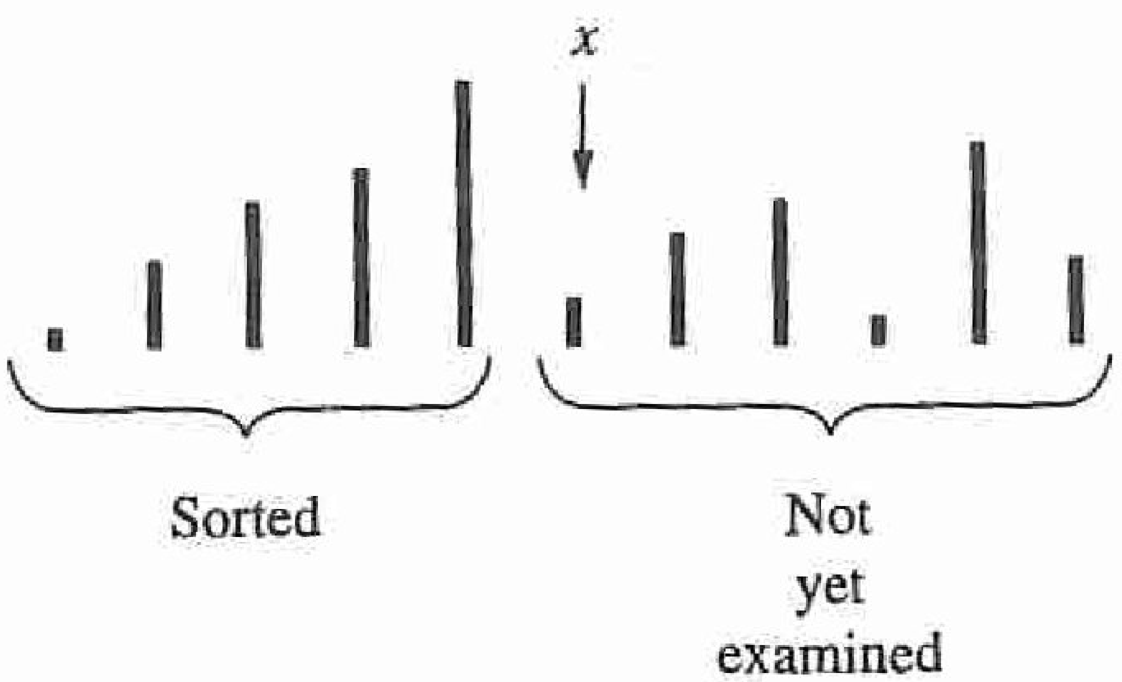

Figure 4.2 Partially sorted elements

Insertion Sort

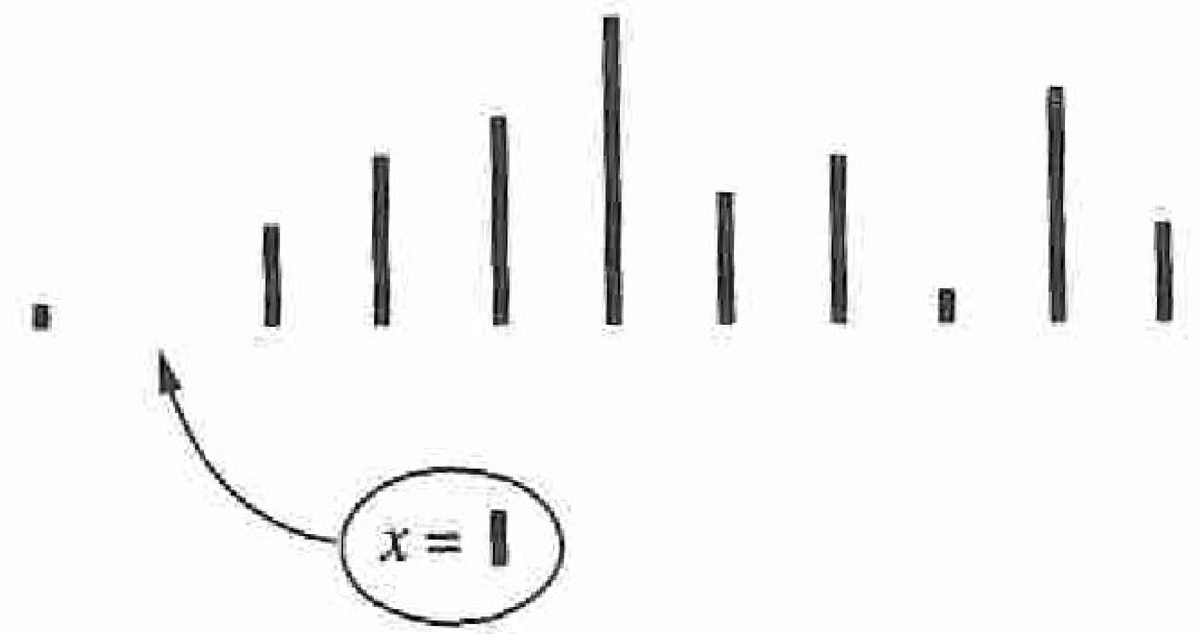

Figure 4.3 Insertion of *x* in proper order

Insertion Sort

```
public static void insertionSort(int[] E)
{
    int n = E.length;
    if (n < 2) return;
    for (int i = 1; i < n; i++)
    {
        int x = E[i];
        E[shiftVacant(E, i, x)] = x;
    }
}
```

Insertion Sort

```java

    private static int shiftVacant(int[] E, int vacant, int x)
{
        while ((vacant > 0) && ((E[vacant - 1] > x) & Bcnt.incr()))
        {
                E[vacant] = E[vacant - 1];
                Bcnt2.incr();
                vacant --;
        }
        return vacant;
}
```

Insertion Sort

Main results

Insertion Sort Worst Case

$$T(n) = \frac{1}{2} n(n-1)$$

Insertion Sort Avg Case

$$T_{avg}(n) \approx \frac{1}{4} n (n + 3) - \text{Log}[n] - 0.577216$$

Bounds in Class

C – a class of sorting algorithms that sort by comparisons of keys and remove at most one inversion after each comparison

Upper Bounds in Class

InsertionSort provides upper bounds for worst-case and average-case number of comparisons in class C.

Worst-case Lower Bound

Therorem 1. Every algorithm in class C must perform at least $\frac{n(n-1)}{2}$ comparisons in the worst case.

Average-case Lower Bound

Therorem 2. Every algorithm in class C must perform at least $\frac{n(n-1)}{4}$ comparisons in the average case.

Average-case Lower Bound +

Theorem. While sorting based only on comparisons of keys, each algorithm in class C must perform at least $\frac{(n-1)(n+2)}{4}$ comparisons.

Insertion Sort Run

```
********* 100 *********
Sorting Array[100]
Array had = 2404 inversions
1 2 10 15 21 24 25 29 34 35 39 40 46 55 59 62 74 100 106 112 116 137 144 162 169 182 186 191 192 215 219 2
comps(100) = 2502
shifts(100) = 2404
comps(100) - inversions(100) = 98
N - Math.log(N)- 0.577216 = 94.8176138140119
Sorting ReversedArray[100]
ReversedArray had = 2546 inversions
1 2 10 15 21 24 25 29 34 35 39 40 46 55 59 62 74 100 106 112 116 137 144 162 169 182 186 191 192 215 219 2
comps(100) = 2642
shifts(100) = 2546
comps(100) - inversions(100) = 96
  4950 = N*(N-1)/2
  5049 = (N+2)*(N-1)/2
  2475 = N*(N-1)/4
Comparisons in both: 5144
Theoretic average number of comps over all arrays of size 100 for InsertionSort = 2569.812613814012
2 * Theoretic average number of comps over all arrays of size 100 for InsertionSort = 5139.625227628024
BUILD SUCCESSFUL (total time: 1 second)
```

Insertion Sort Run

```
********* 100 *********
Sorting Array[100]
Array had = 0 inversions
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 3
comps(100) = 99
shifts(100) = 0
comps(100) - inversions(100) = 99
N - Math.log(N)- 0.577216 = 94.8176138140119
Sorting ReversedArray[100]
ReversedArray had = 4950 inversions
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 3
comps(100) = 4950
shifts(100) = 4950
comps(100) - inversions(100) = 0
  4950 = N*(N-1)/2
  5049 = (N+2)*(N-1)/2
  2475 = N*(N-1)/4
Comparisons in both: 5049
Theoretic average number of comps over all arrays of size 100 for InsertionSort = 2569.812613814012
2 * Theoretic average number of comps over all arrays of size 100 for InsertionSort = 5139.625227628024
BUILD SUCCESSFUL (total time: 1 second)
```

Sorting