CmSc 250 Intro to Algorithms

Sorting Algorithms

1. Assumptions

1. Elements to sort are placed in arrays of length N.
2. Can be compared
3. Sorting can be performed in main memory

Simple sorting algorithms: \( O(N^2) \)
Shell sort: \( o(N^2) \)
Advanced sorting algorithms: \( O(N\log N) \)
In general: \( \Omega(N\log N) \)

2. Insertion Sort

PRE: array of \( N \) elements (from 0 to N-1)
POST: array sorted

1. An array of one element only is sorted
2. Assume that the first \( p \) elements are sorted.
   For \( j = p \) to N-1
   Take the \( j \)-th element and find a place for it among the first \( j \) sorted elements

```c
int j, p;
comparable tmp;
for ( p = 1; p < N ; p++)
{
    tmp = a[p];
    for (j = p; j > 0 && tmp < a[j-1]; j--)
        a[j] = a[j-1];
    a[j] = tmp;
}
```
Analysis of the Insertion Sort

To insert the last element we need at most N-1 comparisons and N-1 movements.
To insert the N-1\textsuperscript{st} element we need N-2 comparisons and N-2 movements.
....
To insert the 2\textsuperscript{nd} element we need 1 comparison and one movement.

To sum up:

\[2 \times (1 + 2 + 3 + \ldots + N - 1) = 2 \times N(N-1)/2 = N(N-1) = \Theta(N^2)\]

if the greater part of the array is sorted, the complexity is almost O(N)
The average complexity is proved to be \(= \Theta(N^2)\)

3. A lower bound for simple sorting algorithms

Simple sorting algorithms swap elements that are not ordered. Swapping is done by bubble sort, and by insertion sort. Thus the complexity depends on the number of swaps. To estimate how many swaps are needed on average, we define inversion in the following way:

Definition 1 An inversion is an ordered pair \((A_i, A_j)\) such that \(i < j\) but \(A_i > A_j\).

Example: 10,6, 7, 15, 3,1
Inversions are: (10,6), (10,7), (10,3),(10,1)  
(6,3), (6,1)  
(7,3), (7,1)  
(15,3), (15,1)  
(3,1)

The following is true:

- Swapping adjacent elements that are out of order removes one inversion.
- A sorted array has no inversions.
- Sorting an array that contains \(i\) inversions requires at least \(i\) (implicit) swaps of adjacent elements.

How many inversions are there in an average unsorted array?
In general this is a tricky question to answer - just what is meant by average? However, if we make a couple of simplifying assumptions:

1. There are no duplicates in the list.
2. Since the elements are unique (by assumption), all that matters is their relative rank. Accordingly we identify them with the first N integers \(\{1, 2, ..., N\}\) and assume the elements we have to sort are the first N integers.
Under these circumstances we can say the following:

**Theorem 1** [Average number of inversions] The average number of inversions in an array of \( N \) distinct elements is

\[
N \frac{(N - 1)}{4}
\]

**Proof:** Given an array \( A \), consider \( A_r \), which is the array in reverse order. Now consider a pair \((x, y)\) with \(x < y\). This pair is an inversion in exactly one of \( A, A_r \). The total number of such pairs is given by \( N \frac{(N - 1)}{2} \), and (on average) half of these will be inversions in \( A \).

Thus \( A \) has \( N \frac{(N - 1)}{4} \) inversions.

Consequently the insertion sort has an average running time of \( O(N^2) \). In fact we can generalize this result to all sorting algorithms that work by exchanging adjacent elements to eliminate inversions.

**Theorem 2** Any algorithm that sorts by exchanging adjacent elements requires \( \Omega(N^2) \) time on average.

The proof follows immediately from the fact that the average number of inversions is \( N(N-1)/4 \): each adjacent swap removes only one inversion, so \( \Omega(N^2) \) swaps are required.

Theorem 2 above implies that for a sorting algorithm to run in less than quadratic time it must do something other than swap adjacent elements.