Introduction to Algorithms

George Voutsadakis[1]

[1]Mathematics and Computer Science
Lake Superior State University

LSSU Math 400

Subsection 1

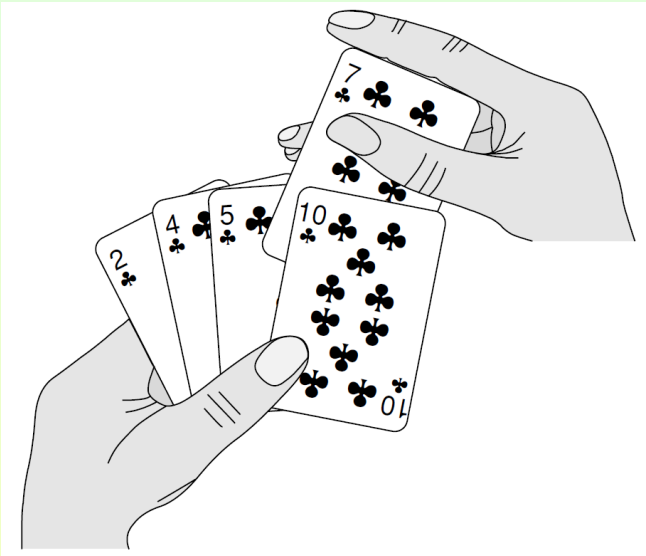
The Algorithm

The Sorting Problem

- Insertion sort sorts n given numbers:
  - **Input**: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.
  - **Output**: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.
- The numbers that we wish to sort are also known as the **keys**.
- Although conceptually we are sorting a sequence, the input comes to us in the form of an array with n elements.

Idea Behind Insertion Sort

- Insertion sort is an efficient algorithm for sorting a small number of elements and works in the same way we sort a hand of playing cards.



- We start with an empty left hand and the cards face down on the table.
- We then remove one card at a time from the table and insert it into the correct position in the left hand.
- To find the correct position for a card, we compare it with each of the cards already in the hand, from right to left.
- At all times, the cards held in the left hand are sorted, and these cards were originally the top cards of the pile on the table.

The procedure InsertionSort

- It takes as a parameter an array $A[1 \ldots n]$ containing a sequence of length n that is to be sorted.

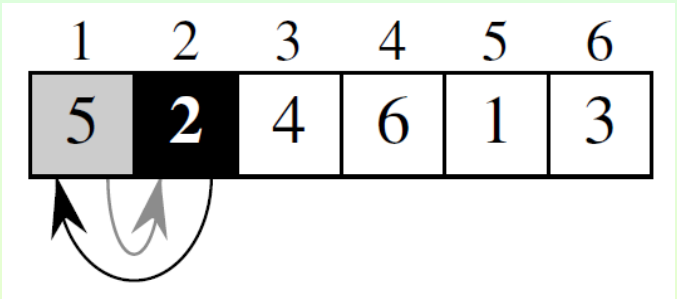
  The number n of elements in A is denoted by A.length.
- The input numbers are sorted in place: The numbers are rearranged within the array A, with at most a constant number of them stored outside the array at any time.
- At the end, the input array A contains the sorted output.

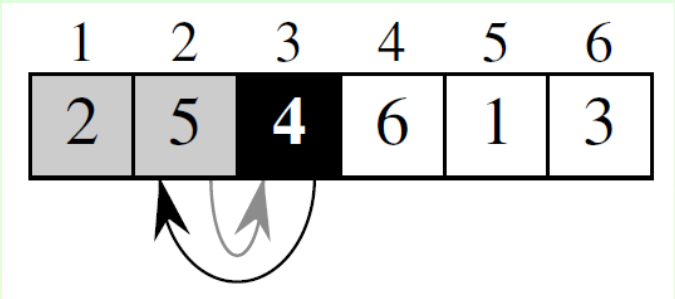
InsertionSort(A)

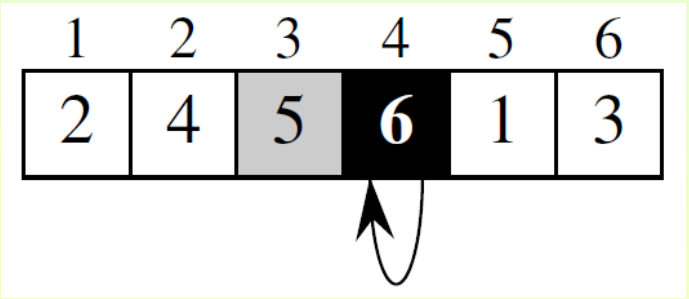
```
1. for j = 2 to A.length
2.     key = A[j]
3.     // Insert A[j] into the sorted sequence A[1 ... j − 1].
4.     i = j − 1
5.     while i > 0 and A[i] > key
6.         A[i + 1] = A[i]
7.         i = i − 1
8.     A[i + 1] = key
```

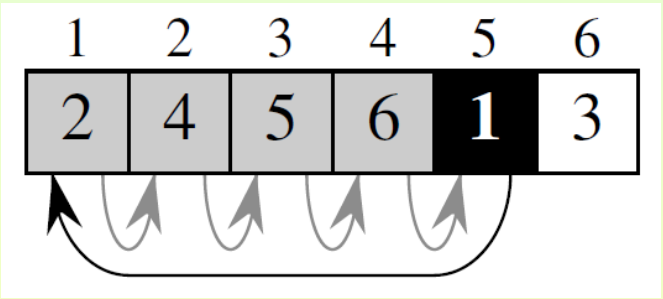
Illustration of Insertion Sort

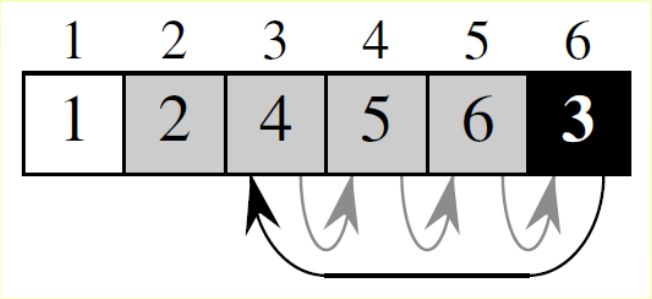
- We illustrate how the algorithm works for $A = \langle 5, 2, 4, 6, 1, 3 \rangle$.

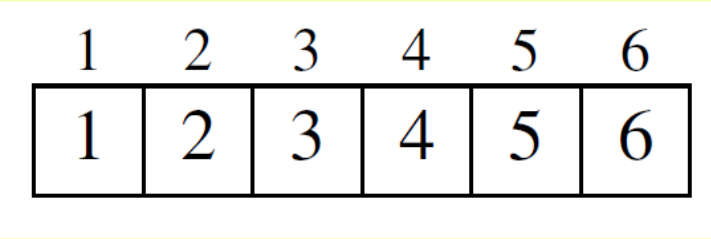












Subsection 2

Correctness of the Algorithm

The Loop Invariant

- The index j indicates the "current card" being inserted into the hand.
  - At the beginning of each iteration of the for loop, which is indexed by j, the subarray consisting of elements $A[1 \ldots j-1]$ constitutes the currently sorted hand.
  - The remaining subarray $A[j+1 \ldots n]$ corresponds to the pile of cards still on the table.
  - The elements $A[1 \ldots j-1]$ are the elements originally in positions 1 through $j-1$, but now in sorted order.
- We state these properties of $A[1 \ldots j-1]$ formally as a loop invariant:

  At the start of each iteration of the for loop of Lines 1-8, the subarray $A[1 \ldots j-1]$ consists of the elements originally in $A[1 \ldots j-1]$, but in sorted order.

Using the Loop Invariant for Correctness

- We must show three things about a loop invariant:
  - **Initialization**: It is true prior to the first iteration of the loop.
  - **Maintenance**: If it is true before an iteration of the loop, it remains true before the next iteration.
  - **Termination**: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.
- When the first two properties hold, the loop invariant is true prior to every iteration of the loop.
- By analogy with mathematical induction:
  - The invariant holding before the first iteration corresponds to the base case;
  - Showing that the invariant holds from iteration to iteration corresponds to the inductive step.
- The third property is the most important one, since we are using the loop invariant to show correctness.

Initialization and Maintenance for Insertion Sort

- Let us see how these properties hold for insertion sort:
  - **Initialization**: We start by showing that the loop invariant holds before the first loop iteration, when $j = 2$. The subarray $A[1 \ldots j-1]$, therefore, consists of just the single element $A[1]$. This is in fact the original element in $A[1]$. Moreover, this subarray is sorted. This shows that the loop invariant holds prior to the first iteration of the loop.
  - **Maintenance**: Next, we show that each iteration maintains the loop invariant. Informally, the body of the for loop works by moving $A[j-1]$, $A[j-2]$, $A[j-3]$, and so on by one position to the right until it finds the proper position for $A[j]$ (Lines 4-7), at which point it inserts the value of $A[j]$ (Line 8). The subarray $A[1 \ldots j]$ then consists of the elements originally in $A[1 \ldots j]$, but in sorted order. Incrementing j for the next iteration of the for loop then preserves the loop invariant.
    A more formal treatment of the second property would require us to state and show a loop invariant for the while loop of Lines 5-7.

Termination and Correctness for Insertion Sort

- **Termination**: Finally, we examine what happens when the loop terminates. The condition causing the for loop to terminate is that $j > \text{A.length} = n$. Because each loop iteration increases j by 1, we must have $j = n + 1$ at that time. Substituting $n + 1$ for j in the wording of loop invariant, we have that the subarray $\text{A}[1 \ldots n]$ consists of the elements originally in $\text{A}[1 \ldots n]$, but in sorted order. Observing that the subarray $\text{A}[1 \ldots n]$ is the entire array, we conclude that the entire array is sorted.

Hence, the algorithm is correct.

Subsection 3

Analysis of the Algorithm

Size of Input and Running Time

- We define **input size** depending on the problem:
  - For many problems, such as sorting, the most natural measure is the number of items in the input.
  - For many other problems, such as multiplying two integers, the best measure of input size is the total number of bits needed to represent the input in ordinary binary notation.
  - Sometimes, it is more appropriate to describe the size of the input with two numbers rather than one. For instance, if the input to an algorithm is a graph, the input size can be described by the numbers of vertices and edges in the graph.
- The **running time** of an algorithm on a particular input is the number of primitive operations or "steps" executed.
  - A constant amount of time is required to execute each line of our pseudocode.
  - One line may take a different amount of time than another line, but we assume that each execution of the i-th line takes time c_i, where c_i is a constant.

InsertionSort(A) With Costs and Times of Execution

- We present InsertionSort with the time "cost" of each statement and the number of times each statement is executed:
  - For each $j = 2, \ldots, n$, where $n =$ A.length, let t_j be the number of times the while loop test in Line 5 is executed for that value of j.

InsertionSort(A) With Costs and Times

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j - 1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 $A[i+1] = key$	c_8	$n-1$

Running Time: The Best Case

- The running time of the algorithm is the sum of running times for each statement executed: a statement that takes c_i steps to execute and executes n times will contribute $c_i n$ to the total running time.
- The running time $T(n)$ of InsertionSort on an input of n values is:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j \\ &\quad + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1). \end{aligned}$$

- In InsertionSort, the best case occurs if the array is already sorted: For each $j = 2, 3, \ldots, n$, we then find that $A[i] \leq$ key in Line 5 when i has its initial value of $j - 1$. Thus, $t_j = 1$ for $j = 2, 3, \ldots, n$. So the best-case running time is

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8). \end{aligned}$$

We can express this running time as $an + b$, for constants a and b that depend on the statement costs c_i. This is a linear function of n.

Running Time: The Worst Case

- If the array is in decreasing order the worst case results. We must compare each element A[j] with each element in the entire sorted subarray A[$1 \ldots j-1$]. So $t_j = j$, for $j = 2, 3, \ldots, n$. Note $\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$ and $\sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$. Thus, in the worst case, the running time of InsertionSort is

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\frac{n(n-1)}{2} + c_7\frac{n(n-1)}{2} + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 \\ &\quad + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8). \end{aligned}$$

We can express this worst-case running time as $an^2 + bn + c$ for constants a, b and c that again depend on the statement costs c_i, i.e., a quadratic function of n.

Worst-Case and Average-Case Analysis

- We usually concentrate on finding only the worst-case running time for any input of size n:
  - The worst-case running time of an algorithm gives us an upper bound on the running time for any input.
  - For some algorithms, the worst case occurs fairly often.
  - The "average case" is often roughly as bad as the worst case.
- Suppose that we randomly choose n numbers and apply insertion sort. On average, half the elements in $A[1 \ldots j-1]$ are less than $A[j]$, and half the elements are greater. On average, therefore, we check half of the subarray $A[1 \ldots j-1]$. So t_j is about $\frac{j}{2}$. The resulting average-case running time turns out to be a quadratic function of the input size, just like the worst-case running time.

Order of Growth

- We used some simplifying abstractions to ease our analysis of the INSERTIONSORT procedure.
  - Instead of actual costs, we used the c_i's to represent them.
  - Even these constants give us more detail than we really need:
    - We expressed the worst-case running time as $an^2 + bn + c$, for some constants a, b and c that depend on the statement costs c_i.
    - We, thus, eventually ignored even the abstract costs c_i.
- We simplify further by assuming we are only interested in the rate of growth, or order of growth, of the running time.
  - We therefore consider only the leading term of a formula (an^2), since the lower-order terms are relatively insignificant for large values of n.
  - We also ignore the leading term's constant coefficient, since constant factors are less significant than the rate of growth in determining computational efficiency for large inputs.

For insertion sort, we are left with the factor of n^2 from the leading term. We write that insertion sort has a worst-case running time of $\Theta(n^2)$.