**Insertion sort**

Our first algorithm, insertion sort, solves the ***sorting problem .***It is an efficient algorithm for sorting a small number of elements, typically up to a few hundred roughly 200 to 900 elements.

The reason insertion sort is considered efficient for small arrays is that its time complexity is O(n^2), which means its performance degrades rapidly as the number of elements increases. For large arrays, more efficient sorting algorithms like quicksort, mergesort, or heapsort, which have an average time complexity of O(n log n), are usually preferred.

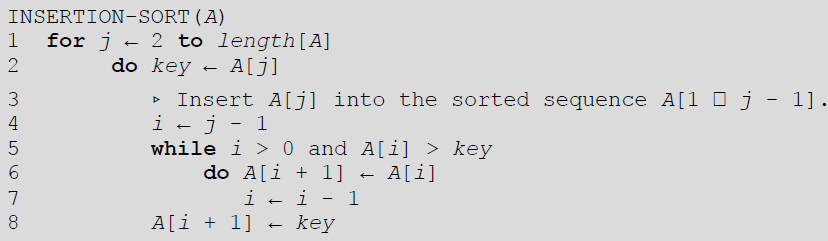
**Input:** A sequence of *n* numbers *a*1, *a*2, . . .,*an*\_.

•.The numbers that we wish to sort are a

lso known as the ***keys***. We will describe algorithm as programs written in pseudocode. Pseudocode is written in clearest method (English) along with language constructs, and concise to specify a given algorithm. Issues of software engineering, modularity and error handling are ignored at Pseudocode level.

Insertion sort takes as a parameter an array *A*[1 \_ *n*] containing a sequence of length *n* that is to be sorted. The input numbers are ***sorted in place***: the numbers are rearranged within the array *A*,

The input array *A* contains the sorted output sequence when INSERTION-SORT is finished.



Example:

****

**Analysis of insertion sort**

The time taken by the INSERTION-SORT procedure depends on the input: sorting a thousand

numbers takes longer than sorting three numbers. Moreover, INSERTION-SORT can take

different amounts of time to sort two input sequences of the same size depending on how

nearly sorted they already are. In general, the time taken by an algorithm grows with the size

of the input, so it is traditional to describe the running time of a program as a function of the size of its input.

The best notion for ***input size*** depends on the problem being studied. For many problems,

such as sorting or computing discrete Fourier transforms, the most natural measure is the

*number of items in the input*-for example, the array size *n* for sorting. 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

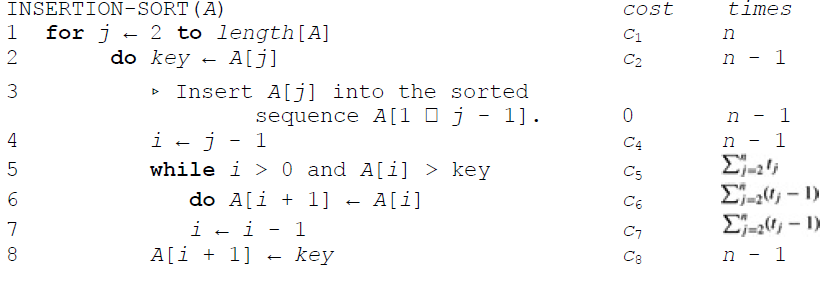
and edges in the graph. The ***running time*** of an algorithm on a particular input is the number of primitive operations or "steps" executed.

We start by presenting the INSERTION-SORT procedure with the time "cost" of each

statement and the number of times each statement is executed. For each *j* = 2, 3, . . . , *n*, where

*n* = *length*[*A*], we let *tj* be the number of times the **while** loop test in line 5 is executed for that

value of *j*.

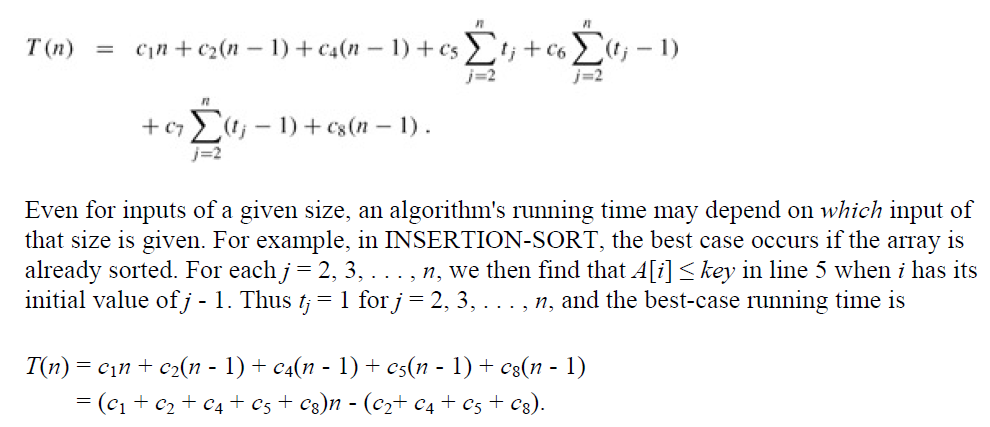


The running time of the algorithm is the sum of running times for each statement executed; a

statement that takes *ci* steps to execute and is executed *n* times will contribute *cin* to the total

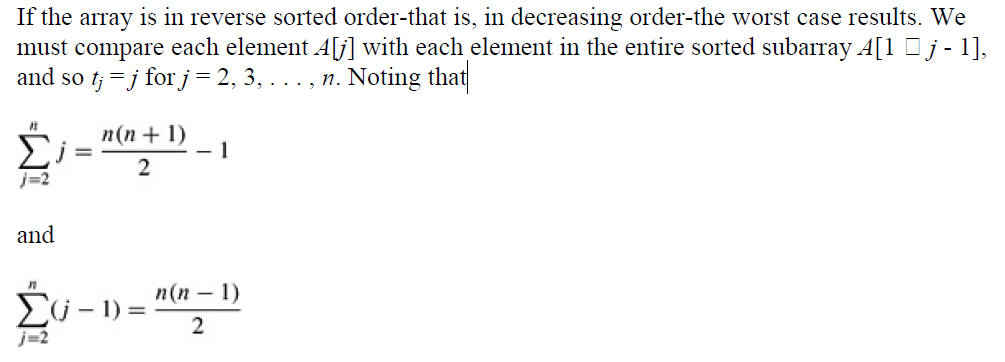
running time. To compute *T*(*n*), the running time of INSERTION-SORT, we sum the

products of the *cost* and *times* columns, obtaining



This running time can be expressed as *an* + *b* for *constants a* and *b* that depend on the

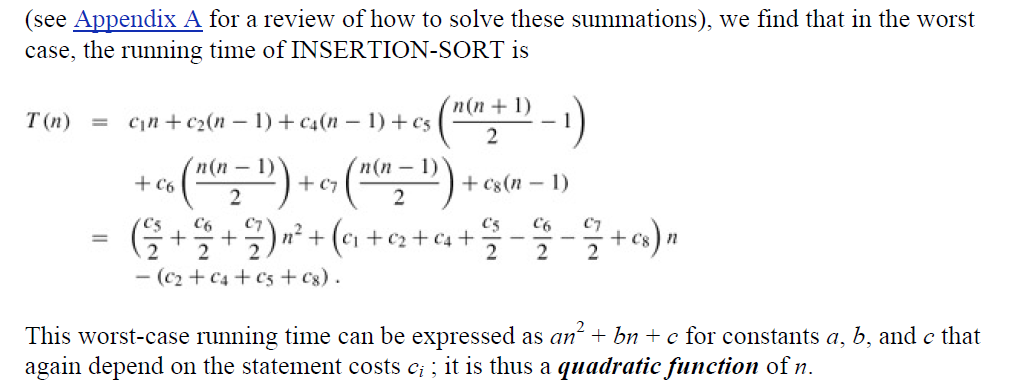
statement costs *ci* ; it is thus a ***linear function*** of *n*.



For j=2, while loop test condition will be executed 2 times.

For j=3, while loop test condition will be executed3 times and so on

For j=n, while loop test condition will be executed n times and so on



**Analysis of insertion sort through Tracing**

**Worst Case Analysis (Usually Done)**  
In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed.

In case of insertion sort algorithm, worst case occurs when the array is in reverse order.

|  |  |  |
| --- | --- | --- |
| j | i | Body Statements (While loop Test condition) (i>0 and A[i] >key)  tj=j |
| 2 | 1 | 2 times t2=2 |
| 3 | 2 | 3 times t3=3 |
| 4 | 3 | 4 times t4=4 |
| How much times? | | |
| n |  | n times tn=n |

Total time taken by while loop test condition is=

Outer most For loop executes n times, so sum up the number of times the body statements (while loops) get executed that is; 2+3+4+…………n= n(n+1)/2 -1

Time complexity= T(n)= Ꝋ(n2)

**Best Case Analysis**

In the Best case analysis, we calculate Lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed.

In case of insertion sort algorithm, Best case occurs when the array is already in sorted order.

|  |  |  |
| --- | --- | --- |
| j | i | Body Statements (While loop Test condition) |
| 2 | 1 | 1 times |
| 3 | 2 | 1 times |
| 4 | 3 | 1 times |
| How much times? | | |
| n |  | 1 times |

= =3 +3+3+3+….3= n. 3= 3n

= =1 +1+1+1+1= 5. 1= 5

= = (n-2+1).1= n-1

= =1 +1+1…….1

Time complexity= T(n)= Ꝋ(n)

= =1 +1+1+1+….1= n. 1= n

