Data Structures & Algorithms

19: Insertion Sort

(Algorithm Analysis)

Dr Ram Prasad Krishnamoorthy

Associate Professor
School of Computing and Data Science

ram.krish@saiuniversity.edu.in

Insertion Sort

Insertion Sort

The sorting problem

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 sequences are typically stored in arrays.

Insertion Sort

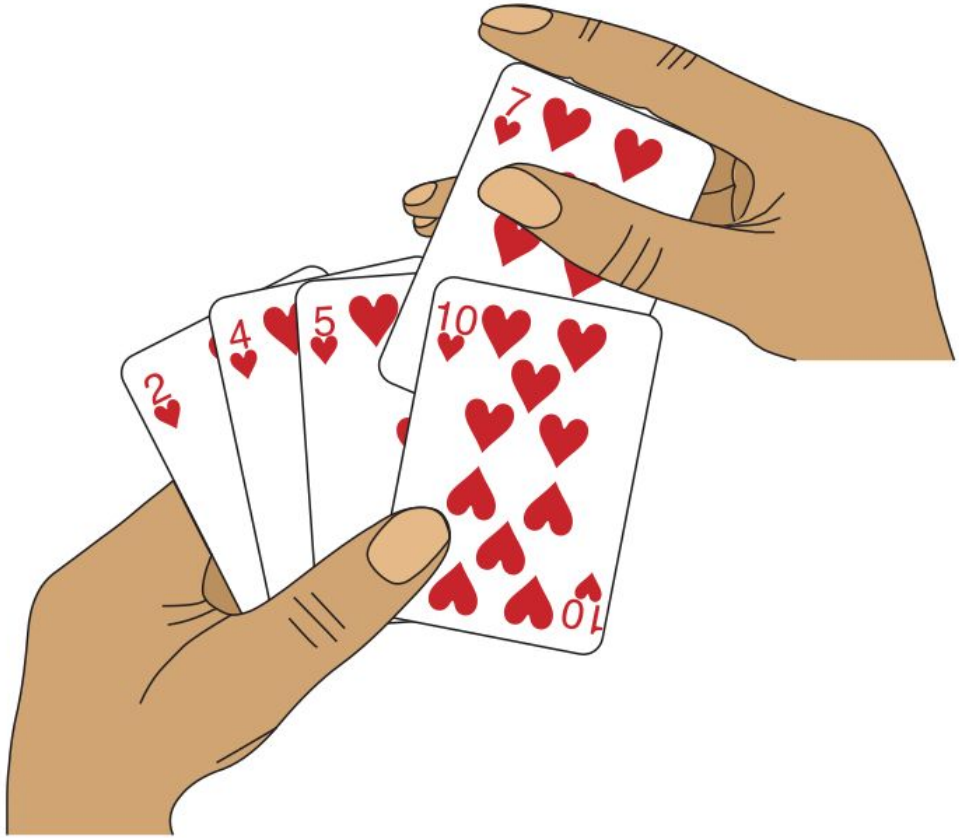
Insertion sort

A good algorithm for sorting a small number of elements.

It works the way you might sort a hand of playing cards:

- Start with an empty left hand and the cards face down on the table.
- 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, 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.

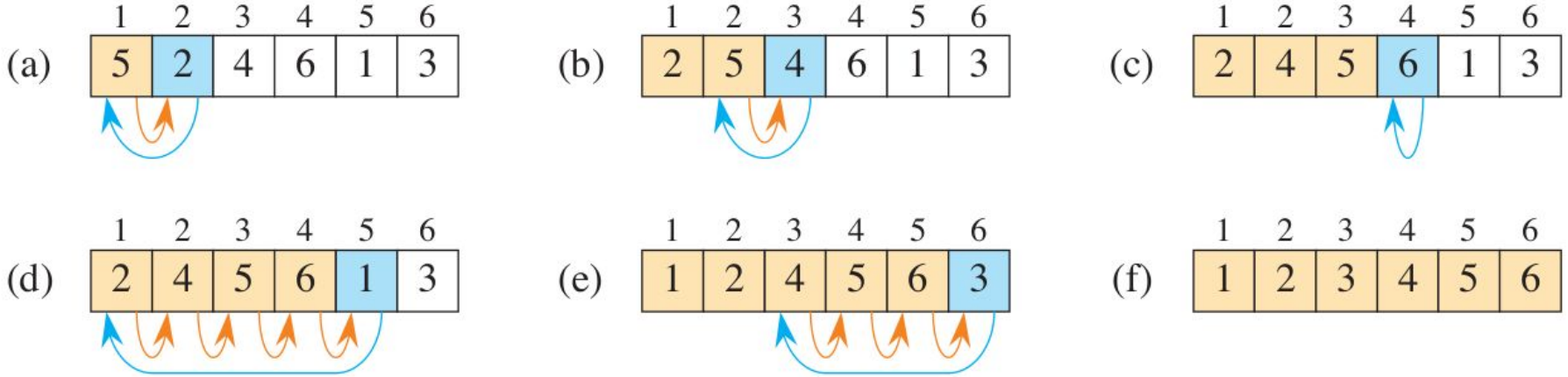
Insertion Sort



Insertion Sort

```
INSERTION-SORT(A, n)
  for i = 2 to n
      key = A[i]
      // Insert A[i] into the sorted subarray A[1 : i − 1].
      j = i − 1
      while j > 0 and A[j] > key
          A[j + 1] = A[j]
          j = j − 1
      A[j + 1] = key
```

Insertion Sort



Analyzing Algorithms

Analyzing Algorithms

Algorithm Analysis

Estimating the resources needed for the completion of an algorithmic task.

- **Time complexity** - runtime required for completion of the task.
- **Space complexity** - memory required for completion of the task.

Time complexity analysis is more prominently used.

A generic **one-processor Random Access Memory** model is used for analysis.

- Sequential computation of **instructions** is assumed.
- Each instruction takes a **constant** amount of **time**.

Analysis based on input size

Analyzing Algorithms

How do we analyze an algorithm's running time?

The time taken by an algorithm depends on the input.

- Sorting 1000 numbers takes longer than sorting 3 numbers.
- A given sorting algorithm may even take differing amounts of time on two inputs of the same size.
- For example, we'll see that insertion sort takes less time to sort n elements when they are already sorted than when they are in reverse sorted order.

INPUT SIZE

Analyzing Algorithms

Input size

Depends on the problem being studied.

- Usually, the number of items in the input. Like the size n of the array being sorted.
- But could be something else. If multiplying two integers, could be the total number of bits in the two integers.
- Could be described by more than one number. For example, graph algorithm running times are usually expressed in terms of the number of vertices and the number of edges in the input graph.

Insertion Sort Analysis

Insertion Sort

INSERTION-SORT(A, n)	*cost*	*times*
for $i = 2$ **to** n	c_1	n
$key = A[i]$	c_2	$n-1$
// Insert $A[i]$ into the sorted subarray $A[1:i-1]$.	0	$n-1$
$j = i-1$	c_4	$n-1$
while $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
$A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n} (t_i - 1)$
$j = j-1$	c_7	$\sum_{i=2}^{n} (t_i - 1)$
$A[j+1] = key$	c_8	$n-1$

Analyzing Algorithms

Analysis of insertion sort

- Assume that the kth line takes time c_k, which is a constant. (Since the third line is a comment, it takes no time.)
- For $i = 2, 3, \ldots, n$, let t_i be the number of times that the **while** loop test is executed for that value of i.
- Note that when a **for** or **while** loop exits in the usual way—due to the test in the loop header—the test is executed one time more than the loop body.

The running time of the algorithm is

$$\sum_{\text{all statements}} (\text{cost of statement}) \cdot (\text{number of times statement is executed}) .$$

Insertion Sort

INSERTION-SORT(A, n)	*cost*	*times*
for $i = 2$ **to** n	c_1	n
$key = A[i]$	c_2	$n-1$
// Insert $A[i]$ into the sorted subarray $A[1:i-1]$.	0	$n-1$
$j = i-1$	c_4	$n-1$
while $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
$A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n} (t_i - 1)$
$j = j-1$	c_7	$\sum_{i=2}^{n} (t_i - 1)$
$A[j+1] = key$	c_8	$n-1$

Analyzing Algorithms

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} t_i + c_6 \sum_{i=2}^{n} (t_i - 1)$$
$$+ c_7 \sum_{i=2}^{n} (t_i - 1) + c_8(n-1) .$$

The running time depends on the values of t_i. These vary according to the input.

Best Case Analysis

Insertion Sort

INSERTION-SORT(A, n)	*cost*	*times*
for $i = 2$ **to** n	c_1	n
$key = A[i]$	c_2	$n-1$
// Insert $A[i]$ into the sorted subarray $A[1:i-1]$.	0	$n-1$
$j = i-1$	c_4	$n-1$
while $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
$A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n} (t_i - 1)$
$j = j-1$	c_7	$\sum_{i=2}^{n} (t_i - 1)$
$A[j+1] = key$	c_8	$n-1$

Analyzing Algorithms

Best case

The array is already sorted.

- Always find that $A[i] \leq key$ upon the first time the **while** loop test is run (when $i = i - 1$).
- All t_i are 1.
- Running time is

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

- Can express $T(n)$ as $an + b$ for constants a and b (that depend on the statement costs c_k) $\Rightarrow$ $T(n)$ is a *linear function* of n.

Worst Case Analysis

Analyzing Algorithms

Worst case

The array is in reverse sorted order.

- Always find that $A[i] > key$ in while loop test.
- Have to compare *key* with all elements to the left of the ith position $\Rightarrow$ compare with $i - 1$ elements.
- Since the while loop exits because i reaches 0, there's one additional test after the $i - 1$ tests $\Rightarrow t_i = i$.
- $\sum_{i=2}^{n} t_i = \sum_{i=2}^{n} i$ and $\sum_{i=2}^{n} (t_i - 1) = \sum_{i=2}^{n} (i - 1)$.

Insertion Sort

INSERTION-SORT(A, n)	*cost*	*times*
for $i = 2$ **to** n	c_1	n
$key = A[i]$	c_2	$n-1$
// Insert $A[i]$ into the sorted subarray $A[1:i-1]$.	0	$n-1$
$j = i-1$	c_4	$n-1$
while $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
$A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n} (t_i - 1)$
$j = j-1$	c_7	$\sum_{i=2}^{n} (t_i - 1)$
$A[j+1] = key$	c_8	$n-1$

Analyzing Algorithms

Worst case

The array is in reverse sorted order.

- Always find that $A[i] > key$ in while loop test.
- Have to compare *key* with all elements to the left of the ith position $\Rightarrow$ compare with $i - 1$ elements.
- Since the while loop exits because i reaches 0, there's one additional test after the $i - 1$ tests $\Rightarrow t_i = i$.
- $\sum_{i=2}^{n} t_i = \sum_{i=2}^{n} i$ and $\sum_{i=2}^{n} (t_i - 1) = \sum_{i=2}^{n} (i - 1)$.
- $\sum_{i=1}^{n} i$ is known as an ***arithmetic series***

$$\frac{n(n+1)}{2}$$

Analyzing Algorithms

- Since $\sum_{i=2}^{n} i = \left(\sum_{i=1}^{n} i\right) - 1$, it equals $\frac{n(n+1)}{2} - 1$.
- Letting $l = i - 1$, we see that $\sum_{i=2}^{n}(i-1) = \sum_{l=1}^{n-1} l = \frac{n(n-1)}{2}$.
- Running time is

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

- Can express $T(n)$ as $an^2 + bn + c$ for constants a, b, c (that again depend on statement costs) $\Rightarrow$ $T(n)$ is a *quadratic function* of n.

Average Case Analysis

Worst-case and average-case analysis

We usually concentrate on finding the ***worst-case running time***: the longest running time for *any* input of size n.

Reasons

- The worst-case running time gives a guaranteed upper bound on the running time for any input.
- For some algorithms, the worst case occurs often. For example, when searching, the worst case often occurs when the item being searched for is not present, and searches for absent items may be frequent.

Analyzing Algorithms

- Why not analyze the average case? Because it's often about as bad as the worst case.

  Example: Suppose that we randomly choose n numbers as the input to insertion sort.

  On average, the key in $A[i]$ is less than half the elements in $A[1 : i - 1]$ and it's greater than the other half.
  $\Rightarrow$ On average, the **while** loop has to look halfway through the sorted subarray $A[1 : i - 1]$ to decide where to drop *key*.
  $\Rightarrow t_i \approx i/2$.

  Although the average-case running time is approximately half of the worst-case running time, it's still a quadratic function of n.