Design and Analysis of Algorithm (KCS503)

Insertion Sort and its Analysis

Lecture - 3

A Sorting Problem (Incremental Approach)

Input: A sequence of n numbers $a_1, a_2, . . . , a_n$.

Output: A permutation (reordering) $a_1 , a_2 , . . . , a_n$ of the input sequence such that $a_1 \leq a_2 \leq \cdot \cdot \cdot \leq a_n$.

The sequences are typically stored in arrays.

We also refer to the numbers as ***keys***. Along with each key may be additional information, known as ***satellite data***. (You might want to clarify that .satellite data. does not necessarily come from a satellite!)

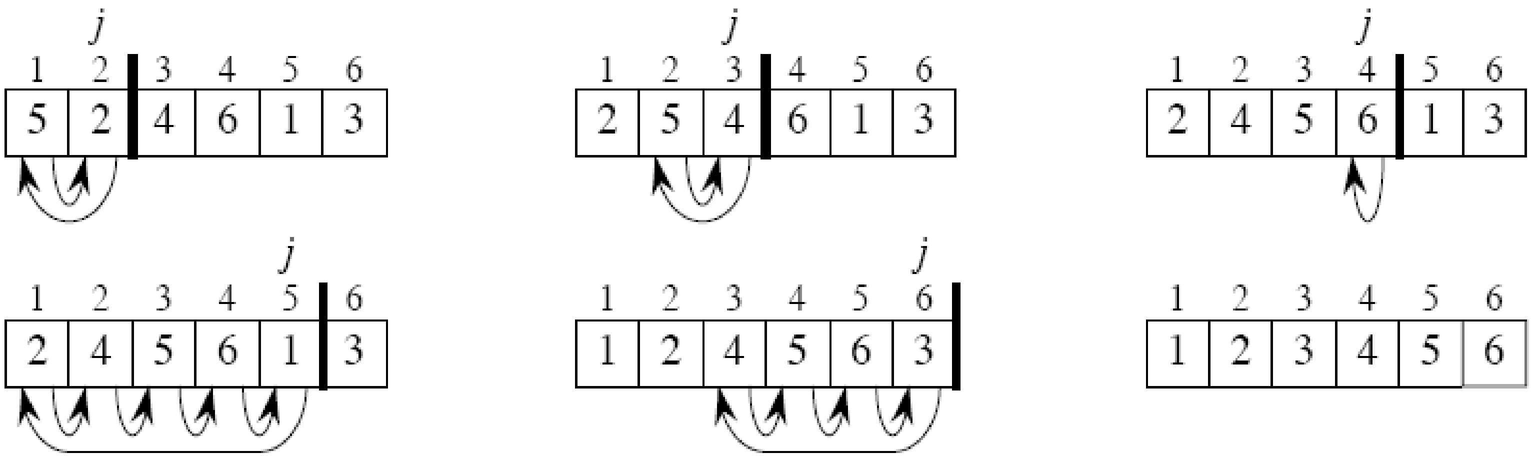
We will see several ways to solve the sorting problem. Each way will be expressed as an ***algorithm***: a well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.

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 (Example)

Insertion sort (Example)



Insertion sort (Algorithm)

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

Correctness Proof of Insertion Sort

- **Initialization:** Just before the first iteration, $j = 2$. The sub array $A[1 \,.\,.\, j - 1]$ is the single element $A[1]$, which is the element originally in $A[1]$, and it is trivially sorted.

- **Maintenance:** To be precise, we would need to state and prove a loop invariant for the "inner" **while** loop. Rather than getting bogged down in another loop invariant, we instead note that the body of the inner **while** loop works by moving $A[j - 1]$, $A[j - 2]$, $A[j - 3]$, and so on, by one position to the right until the proper position for *key* (which has the value that started out in $A[j]$) is found. At that point, the value of *key* is placed into this position.

- **Termination:** The outer **for** loop ends when $j > n$; this occurs when $j = n + 1$. Therefore, $j - 1 = n$. Plugging n in for $j - 1$ in the loop invariant, the sub array $A[1 \,.\,.\, n]$ consists of the elements originally in $A[1 \,.\,.\, n]$ but in sorted order. In other words, the entire array is sorted!

How do we analyze an algorithm's running time?

- ***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.

- ***Running time:*** On a particular input, it is the number of primitive operations (steps) executed.
  - Want to define steps to be machine-independent.
  - Figure that each line of pseudo code requires a constant amount of time.
  - One line may take a different amount of time than another, but each execution of line i takes the same amount of time ci .
  - This is assuming that the line consists only of primitive operations.
    - If the line is a subroutine call, then the actual call takes constant time, but the execution of the subroutine being called might not.
    - If the line specifies operations other than primitive ones, then it might take
    - more than constant time.

Analysis of insertion sort

- Assume that the i th line takes time c_i, which is a constant. (Since the third line is a comment, it takes no time.)
- For $j = 2, 3, . . . , n$, let t_j be the number of times that the **while** loop test is executed for that value of j.
- 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.

Running Time of Insertion Sort

The running time of the algorithm is

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

Let $T(n)$ = running time of INSERTION-SORT.

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

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

Best case: The array is already sorted.

- Always find that $A[i] \le key$ upon the first time the **while** loop test is run (when $i = j - 1$).
- All t_j 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_i) $\Rightarrow T(n)$ is a *linear function* of n.

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 jth position $\Rightarrow$ compare with $j-1$ elements.
- Since the while loop exits because i reaches 0, there's one additional test after the $j-1$ tests $\Rightarrow t_j = j$.
- $\displaystyle\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j$ and $\displaystyle\sum_{j=2}^{n}(t_j - 1) = \sum_{j=2}^{n}(j-1)$.
- $\displaystyle\sum_{j=1}^{n} j$ is known as an ***arithmetic series***, and equation (A.1) shows that it equals $\displaystyle\frac{n(n+1)}{2}$.
- Since $\displaystyle\sum_{j=2}^{n} j = \left(\sum_{j=1}^{n} j\right) - 1$, it equals $\displaystyle\frac{n(n+1)}{2} - 1$.

- Letting $k = j - 1$, we see that $\sum_{j=2}^{n}(j-1) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}$.
- 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.

Home Assignment

- Write the algorithm of **Bubble sort** and calculate the time complexity.