NANYANG TECHNOLOGICAL UNIVERSITY

CE2001/ CZ2001: Algorithms

Insertion Sort

Ke Yiping, Kelly

Learning Objectives

At the end of this lecture, students should be able to:

- Explain the incremental approach as a strategy of algorithm design
- Describe how Insertion sort algorithm works, by manually running its pseudo code on a toy example
- Analyse the time complexities of Insertion sort in the best case, worst case and average case

Insertion Sort of a Hand of Cards

Insertion Sort

The incremental approach

- An intuitive, primitive sorting method
- A form of insertion into an ordered list
- Given an unordered set of objects, repeatedly remove an entry from the set and insert it into a new ordered list
- Ensure that the new list is ordered at all times
- Each insertion requires movements of certain entries in the ordered list

Insertion Sort (Pseudo Code)

The incremental approach

```
void InsertionSort (ALIST slot[ ], int n)
{ // input slot is an array of n records;
  // assume n > 1;
  for (int i=1; i < n; i++)
        for (int j=i; j > 0; j--) {
              if (slot[j].key < slot[j-1].key)
                    swap(slot[j], slot[j-1]);
              else break;
        }
}
```


Insertion Sort (Pseudo Code)

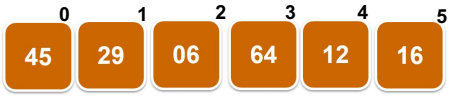
The incremental approach

```
void InsertionSort (ALIST slot[ ], int n)
{ // input slot is an array of n records;
```

Insertion Sort (Pseudo Code)

The incremental approach

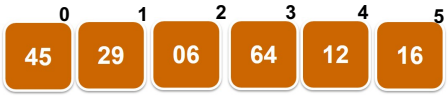
```
void InsertionSort (ALIST slot[ ], int n)
{ // input slot is an array of n records;
```



Insertion Sort (Pseudo Code)

The incremental approach

```
void InsertionSort (ALIST slot[ ], int n)
{ // input slot is an array of n records;
  // assume n > 1;
```



Insertion Sort (Pseudo Code)

The incremental approach

```
void InsertionSort (ALIST slot[ ], int n)
{ // input slot is an array of n records;
  // assume n > 1;
  for (int i=1; i < n; i++)
        for (int j=i; j > 0; j--) {
              if (slot[j].key < slot[j-1].key)
                    swap(slot[j], slot[j-1]);
              else break;
        }
}
```


Insertion Sort (Pseudo Code)

The incremental approach

```
void InsertionSort (ALIST slot[ ], int n)
{ // input slot is an array of n records;
  // assume n > 1;
  for (int i=1; i < n; i++)   Pick up a new item from slot[ ]
        for (int j=i; j > 0; j--) {
              if (slot[j].key < slot[j-1].key)
                    swap(slot[j], slot[j-1]);
              else break;
        }
}
```


Insertion Sort (Pseudo Code)

The incremental approach

```
void InsertionSort (ALIST slot[ ], int n)
{ // input slot is an array of n records;
  // assume n > 1;
  for (int i=1; i < n; i++)
        for (int j=i; j > 0; j--) {   Find the correct position to insert the item.
              if (slot[j].key < slot[j-1].key)
                    swap(slot[j], slot[j-1]);
              else break;
        }
}
```


Insertion Sort (Pseudo Code)

The incremental approach

```
void InsertionSort (ALIST slot[ ], int n)
{ // input slot is an array of n records;
  // assume n > 1;
  for (int i=1; i < n; i++)
        for (int j=i; j > 0; j--) {
              if (slot[j].key < slot[j-1].key)
                    swap(slot[j], slot[j-1]);
              else break;
        }
}
```

Insertion Sort (Pseudo Code)

The incremental approach

```
void InsertionSort (ALIST slot[ ], int n)
{ // input slot is an array of n records;
  // assume n > 1;
  for (int i=1; i < n; i++)
        for (int j=i; j > 0; j--) {
              if (slot[j].key < slot[j-1].key)
                    swap(slot[j], slot[j-1]);
              else break;
        }
}
```


Insertion Sort (Pseudo Code)

The incremental approach

```
void InsertionSort (ALIST slot[ ], int n)
{ // input slot is an array of n records;
  // assume n > 1;
  for (int i=1; i < n; i++)
        for (int j=i; j > 0; j--) {
              if (slot[j].key < slot[j-1].key)
                    swap(slot[j], slot[j-1]);
              else break;
        }
}
```


Insertion Sort (Pseudo Code)

The incremental approach

```
void InsertionSort (ALIST slot[ ], int n)
{ // input slot is an array of n records;
  // assume n > 1;
  for (int i=1; i < n; i++)
        for (int j=i; j > 0; j--) {
              if (slot[j].key < slot[j-1].key)
                    swap(slot[j], slot[j-1]);
              else break;
        }
}
```


Insertion Sort (Pseudo Code)

The incremental approach

```
void InsertionSort (ALIST slot[ ], int n)
{ // input slot is an array of n records;
  // assume n > 1;
  for (int i=1; i < n; i++)
        for (int j=i; j > 0; j--) {
              if (slot[j].key < slot[j-1].key)
                    swap(slot[j], slot[j-1]);
              else break;
        }
}
```

Insertion Sort (Pseudo Code)

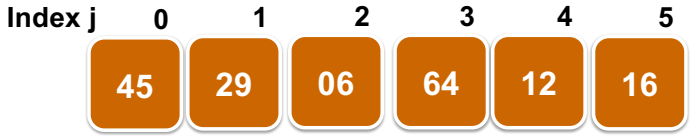
The incremental approach

```
void InsertionSort (ALIST slot[ ], int n)
{ // input slot is an array of n records;
  // assume n > 1;
  for (int i=1; i < n; i++)
        for (int j=i; j > 0; j--) {
              if (slot[j].key < slot[j-1].key)
                    swap(slot[j], slot[j-1]);
              else break;
        }
}
```

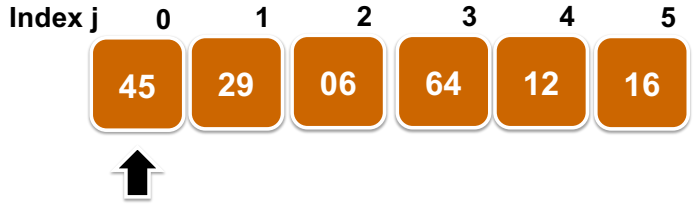

Insertion Sort Example

Insertion Sort Example

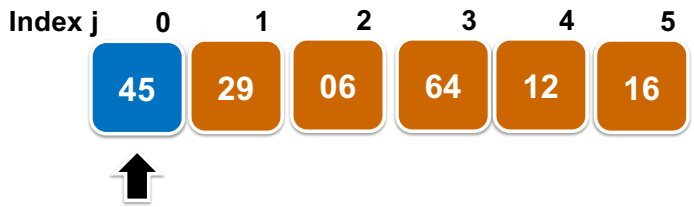
Sort in ascending order



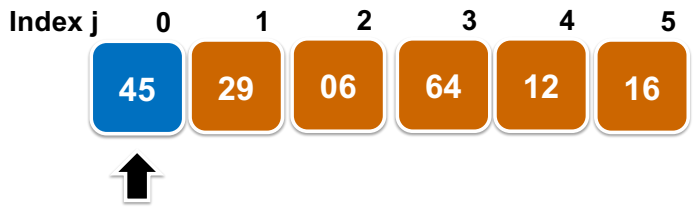
Insertion Sort Example



Insertion Sort Example



Insertion Sort Example

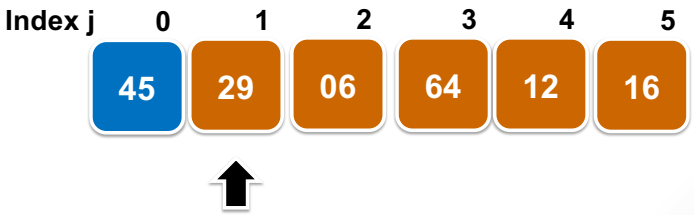


Insertion Sort Example

If (slot[j].key < slot[j-1].key)

(slot[**1**].key < slot[**0**].key)

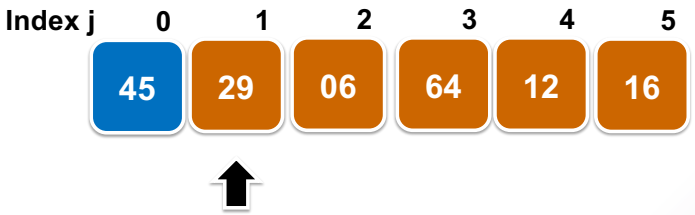
29 < **45** ✓



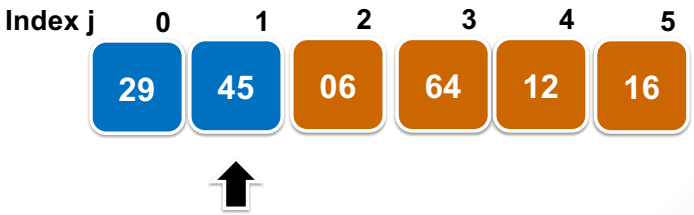
Insertion Sort Example

swap(slot[j], slot[j-1]);

swap(slot[**1**], slot[**0**]);



Insertion Sort Example

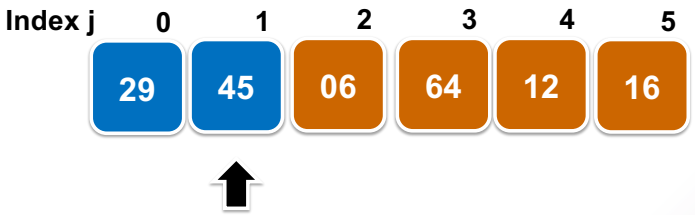


Insertion Sort Example

If (slot[j].key < slot[j-1].key)

(slot[**2**].key < slot[**1**].key)

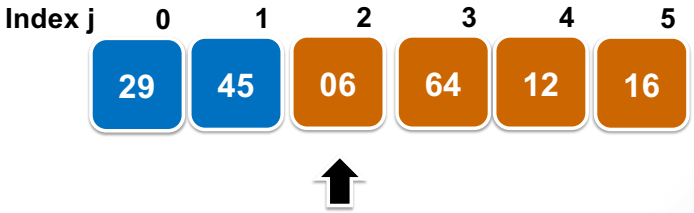
06 < **45** ✓



Insertion Sort Example

swap(slot[j], slot[j-1]);

swap(slot[**2**], slot[**1**]);

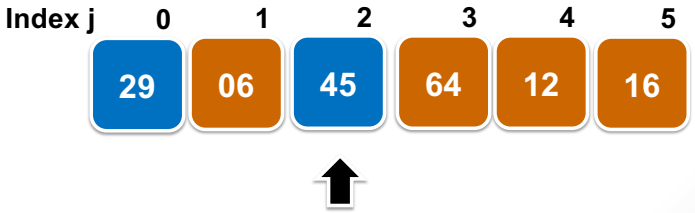


Insertion Sort Example

If (slot[j].key < slot[j-1].key)

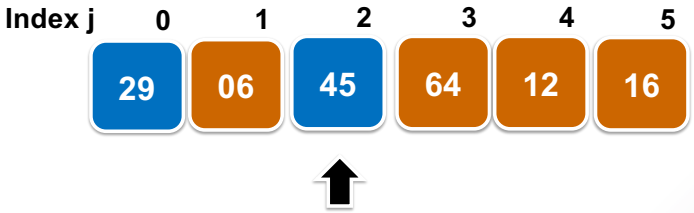
(slot[**1**].key < slot[**0**].key)

06 < **29** ✓

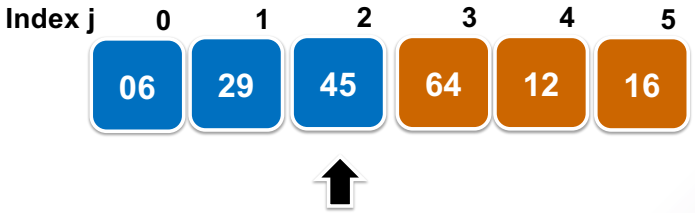


Insertion Sort Example

swap(slot[j], slot[j-1]);
swap(slot[**1**], slot[**0**]);

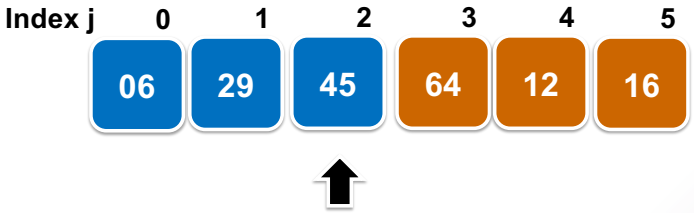


Insertion Sort Example



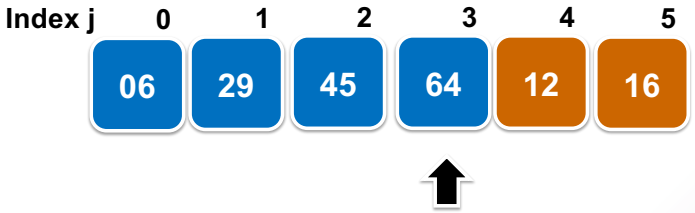
Insertion Sort Example

If (slot[j].key < slot[j-1].key)
(slot[**3**].key < slot[**2**].key)
64 < **45** ✘



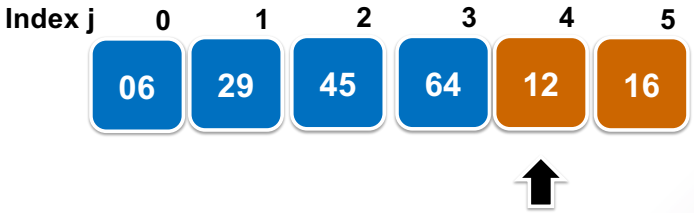
Insertion Sort Example

If (slot[j].key < slot[j-1].key)
(slot[**4**].key < slot[**3**].key)
12 < **64** ✓



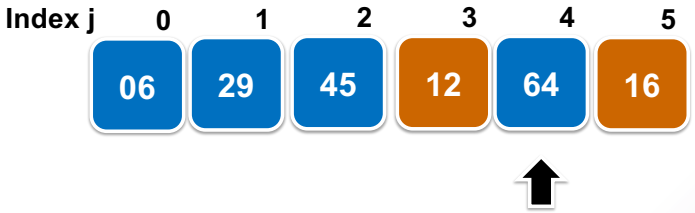
Insertion Sort Example

swap(slot[j], slot[j-1]);
swap(slot[4], slot[3]);



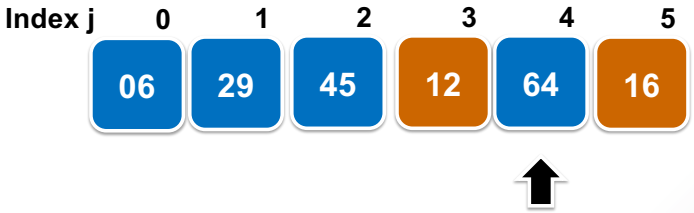
Insertion Sort Example

If (slot[j].key < slot[j-1].key)
(slot[3].key < slot[2].key)
12 < 45 ✓



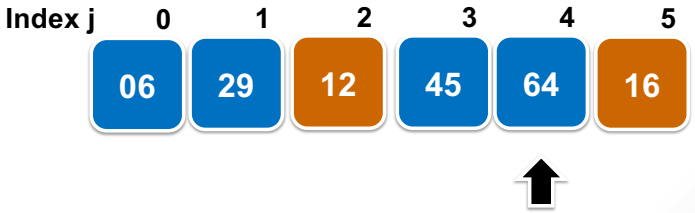
Insertion Sort Example

swap(slot[j], slot[j-1]);
swap(slot[3], slot[2]);



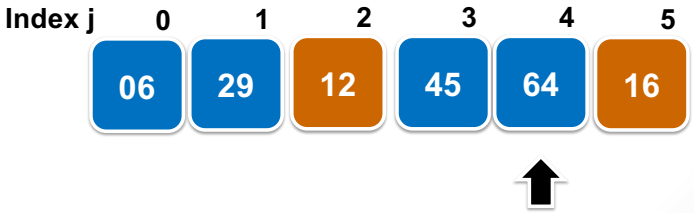
Insertion Sort Example

If (slot[j].key < slot[j-1].key)
(slot[2].key < slot[1].key)
12 < 29 ✓



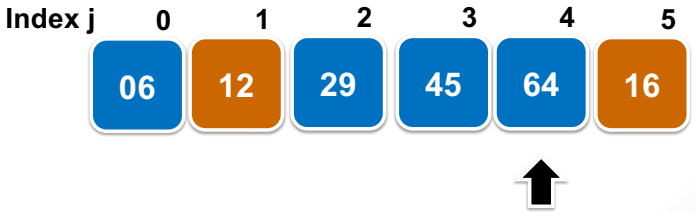
Insertion Sort Example

swap(slot[j], slot[j-1]);
swap(slot[**2**], slot[**1**]);



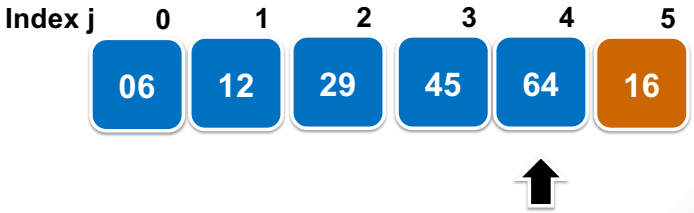
Insertion Sort Example

If (slot[j].key < slot[j-1].key)
(slot[**1**].key < slot[**0**].key)
12 < **06** ✘



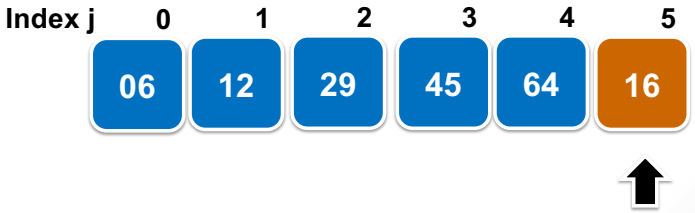
Insertion Sort Example

If (slot[j].key < slot[j-1].key)
(slot[**5**].key < slot[**4**].key)
16 < **64** ✓



Insertion Sort Example

swap(slot[j], slot[j-1]);
swap(slot[**5**], slot[**4**]);

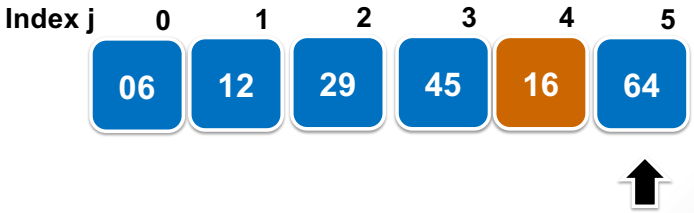


Insertion Sort Example

If (slot[j].key < slot[j-1].key)

(slot[4].key < slot[3].key)

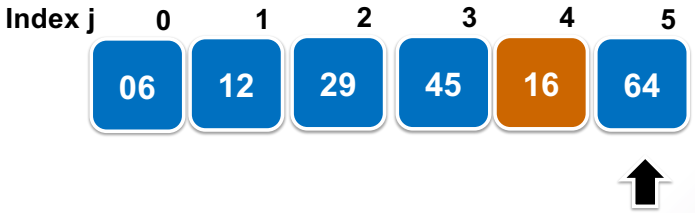
16 < 45 ✓



Insertion Sort Example

swap(slot[j], slot[j-1]);

swap(slot[4], slot[3]);

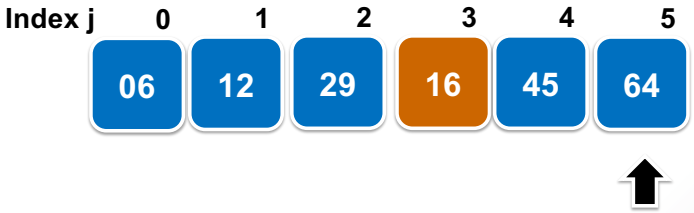


Insertion Sort Example

If (slot[j].key < slot[j-1].key)

(slot[3].key < slot[2].key)

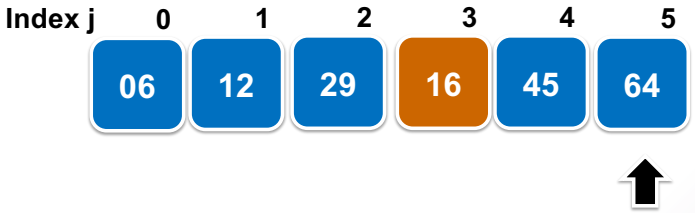
16 < 29 ✓



Insertion Sort Example

swap(slot[j], slot[j-1]);

swap(slot[3], slot[2]);

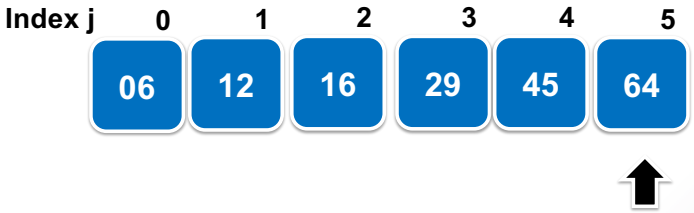


Insertion Sort Example

If (slot[j].key < slot[j-1].key)

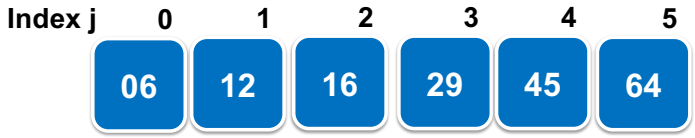
(slot[2].key < slot[1].key)

16 < 12 ✘



Insertion Sort Example

Sorted in ascending order



Insertion Sort Algorithm (Recap)

Insertion Sort Algorithm

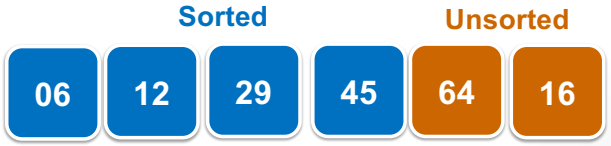
- Original unsorted set and final sorted list are both in array slot[ ].

Insertion Sort Algorithm

- Original unsorted set and final sorted list are both in array slot[ ].
- Since sorting is performed directly on original array without any working storage, swapping and shifting are essential.

Insertion Sort Algorithm

- Original unsorted set and final sorted list are both in array slot[ ].
- Since sorting is performed directly on original array without any working storage, swapping and shifting are essential.
- During sorting, slot[ ] contains sorted portion on the 'left' and unsorted portion on the 'right'; sorted portion grows while unsorted portion shrinks.



Insertion Sort Algorithm

- In the outer 'for' loop, i begins with 1 because the ordered list begins with one element (slot[0]); hence slot[1] is the first element from the unordered list.

```
for (int i=1; i < n; i++)
        for (int j=i; j > 0; j--) {
                if (slot[j].key < slot[j-1].key)
                        swap(slot[j], slot[j-1]);
                else break;
        }
```


Insertion Sort Algorithm

- At each iteration, number at slot[i] is inserted into the new ordered list.

```
for (int i=1; i < n; i++)
        for (int j=i; j > 0; j--) {
                if (slot[j].key < slot[j-1].key)
                        swap(slot[j], slot[j-1]);
                else break;
```

Insertion Sort Algorithm

- The inner 'for' loop finds the correct position in the ordered list by swapping slot[j] with slot[j-1] as long as the key of slot[j-1] is > the key of slot[j].

```
for (int i=1; i < n; i++)
    for (int j=i; j > 0; j--) {
        if (slot[j].key < slot[j-1].key)
            swap(slot[j], slot[j-1]);
        else break;
```


Insertion Sort Algorithm

- The inner 'for' loop finds the correct position in the ordered list by swapping slot[j] with slot[j-1] as long as the key of slot[j-1] is > the key of slot[j].

```
for (int i=1; i < n; i++)
    for (int j=i; j > 0; j--) {
        if (slot[j].key < slot[j-1].key)
            swap(slot[j], slot[j-1]);
        else break;
```


Complexity of Insertion Sort

Complexity of Insertion Sort

Number of key comparisons:

- There are ***n* − 1** iterations **(the outer loop)**
- **Best case:** 1 key comparison/ iteration, **total: *n* − 1**
- **Already sorted:** [06] [12] [16] [29] [45] [64]
- **Worst case:** *i* key comparisons for the *i*th iteration
- **Reversely sorted:** [64] [45] [29] [16] [12] [06]

Total: $1+2+3+\ldots+(n-1)=\sum_{i=1}^{n-1} i=\frac{(n-1)n}{2}$

Insertion Sort Performance

- **Average case:** the *i*th iteration may have 1, 2, …, *i* key comparisons, each with 1/*i* chance.

The average no. of comparisons in the *i*th iteration:

$$\frac{1}{i}\sum_{j=1}^{i} j = \frac{1}{i}(1+2+...+i)$$

Summation for the ***n*-1** iterations:

$$1+\frac{1}{2}(1+2)+\frac{1}{3}(1+2+3)+...+\frac{1}{n-1}(1+...+n-1) = \sum_{i=1}^{n-1}(\frac{1}{i}\sum_{j=1}^{i} j)$$

$$= \sum_{i=1}^{n-1}\left(\frac{1}{i}\frac{i(i+1)}{2}\right) = \frac{1}{2}\sum_{i=1}^{n-1}(i+1) = \frac{1}{2}\left(\frac{(n-1)(n+2)}{2}\right) = \Theta(n^2)$$

Insertion Sort Performance

☺ **Strengths:**

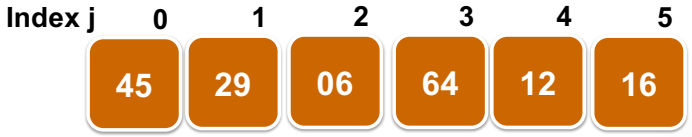
- Good when the unordered list is almost sorted.
- Need minimum time to verify if the list is sorted.
- Fast with linked storage implementation: no movement of data.

☹ **Weaknesses:**

- When an entry is inserted, it may still not be in the final position yet.
- Every new insertion necessitates movements for some inserted entries in ordered list.
- When each slot is large (e.g., a slot contains a large record of 10Mb), movement is expensive.
- Less suitable with contiguous storage implementation.

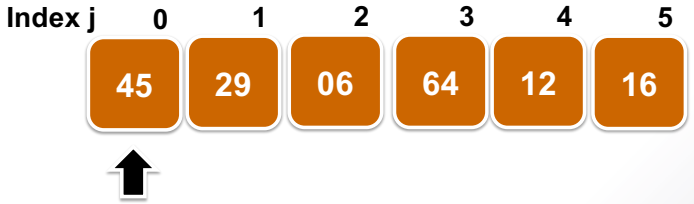
Summary

- Insertion sort uses the incremental approach.
- **Main idea:** Repeatedly pick up an element *x* to insert into a sorted sub-array on the left side, by comparing *x* with its left neighbour. If they are out of order, swap them; otherwise, insert *x* there.



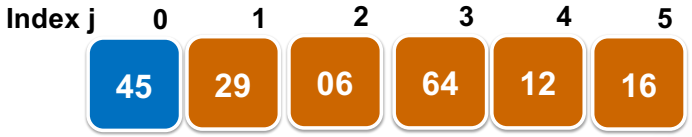
Summary

- Insertion sort uses the incremental approach.
- **Main idea:** Repeatedly pick up an element *x* to insert into a sorted sub-array on the left side, by comparing *x* with its left neighbour. If they are out of order, swap them; otherwise, insert *x* there.



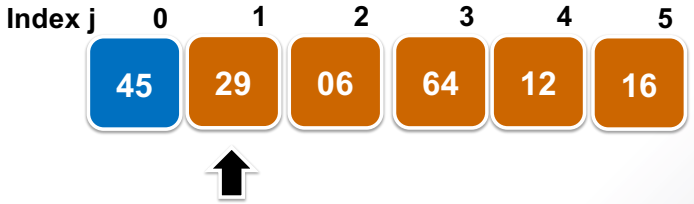
Summary

- Insertion sort uses the incremental approach.
- **Main idea:** Repeatedly pick up an element *x* to insert into a sorted sub-array on the left side, by comparing *x* with its left neighbour. If they are out of order, swap them; otherwise, insert *x* there.



Summary

- Insertion sort uses the incremental approach.
- **Main idea:** Repeatedly pick up an element *x* to insert into a sorted sub-array on the left side, by comparing *x* with its left neighbour. If they are out of order, swap them; otherwise, insert *x* there.



Summary

- Insertion sort uses the incremental approach.
- **Main idea:** Repeatedly pick up an element *x* to insert into a sorted sub-array on the left side, by comparing *x* with its left neighbour. If they are out of order, swap them; otherwise, insert *x* there.
- **Time complexity analysis:**
  - Best case: $\Theta(n)$, when input array is already sorted.
  - Worst case: $\Theta(n^2)$, when input array is reversely sorted.
  - Average case: $\Theta(n^2)$.