



CE2101/ CZ2101: Algorithm Design and Analysis

Insertion Sort

Ke Yiping, Kelly

复制的摘要 slide

desired outcome: 1 2 3 4 5 6.

Best Case: input 1 2 3 4 5 6.

 $O(n)$ time = $n-1$ (每个项比较一次)

Worst Case: input 6 5 4 3 2 1

 $O(n^2)$ time = $\sum_{i=2}^n (i-1) = 1+2+3+\dots+n-1$ (每项比较 $i-1$ 次)
 $= \frac{(1+n-1) \times (n-1)}{2} = \frac{n(n-1)}{2}$

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;

}

}

* Average Case: input 3 5 6 2 1 4

 $O(n^2)$ 每一项都可能比较 1, 2, ..., i 次, 比较次数出现的几率都是 $\frac{1}{i}$.

对任意一项, $t = \sum_{j=1}^i \frac{1}{i} \times j = \frac{1}{i} \sum_{j=1}^i j = \frac{1}{i} \times \frac{i(i+1)}{2} = \frac{i+1}{2}$

对全部项: $t = \sum_{i=1}^{n-1} \frac{i+1}{2} = \frac{(1 + \frac{n+1}{2}) \times n}{2} = \frac{n(n+1)}{4}$ $O(n^2)$

Learning Objectives

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

- Explain the incremental approach as a strategy of algorithm design
sort one-by-one
- 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

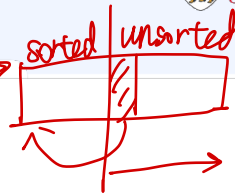
- An intuitive, primitive sorting method

- 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



Sorted: grow by 1
unsorted: shrink by 1

Insertion Sort (Pseudo Code)

The incremental approach

```
void InsertionSort (arrayALIST size.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;
```

```
    }
```

```
}
```

45

29

06

64

12

16

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++)
```

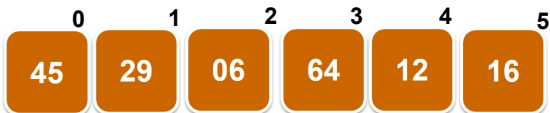
```
    for (int j=i; j > 0; j--) {  
        if (slot[j].key < slot[j-1].key)
```

```
            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--) {
```

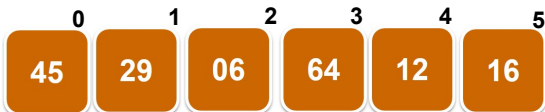
```
      if (slot[j].key < slot[j-1].key)
```

```
        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  $n > 1$ ; → index starts at 0.
```

```
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--) {
```

```
      if (slot[j].key < slot[j-1].key)
```

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

```
      else break;
```

```
    }
```

```
}
```

→ inner loop.

往前查找位置

Find the correct position to insert the item.

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;
```

```
    }
```

```
}
```

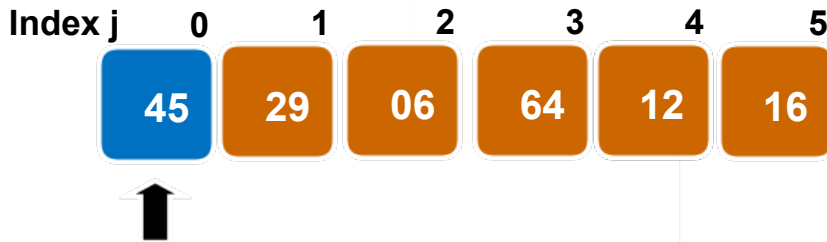
compare only key values.

swap record.

*该算法 swap 了多次。

Insertion Sort Example

Insertion Sort Example



Insertion Sort Example

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

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

16 < 12 ☐

Index j	0	1	2	3	4	5
		12	16	29	45	64



Insertion Sort Algorithm (Recap)

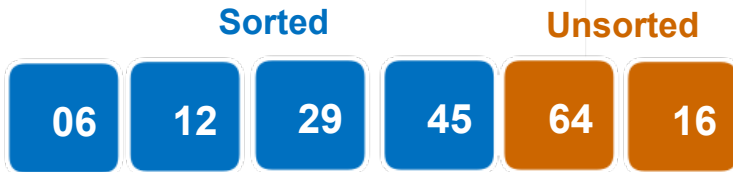
Insertion Sort Algorithm

- 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 *No extra space required* 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 ($\text{slot}[0]$); hence $\text{slot}[1]$ 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.

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]

$$\text{Total: } 1 + 2 + 3 + \dots + (n - 1) = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \quad \leftarrow \Theta(n^2)$$

Insertion Sort Performance

Average case: the i th iteration may have $1, 2, \dots, i$ key comparisons, each with $1/i$ chance.

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

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

Summation for the $n-1$ iterations:

$$\begin{aligned} 1 + \frac{1}{2}(1+2) + \frac{1}{3}(1+2+3) + \dots + \frac{1}{n-1}(1+\dots+n-1) &= \sum_{i=1}^{n-1} \left(\frac{1}{i} \sum_{j=1}^i j \right) \\ &= \sum_{i=1}^{n-1} \left(\frac{1}{i} \frac{i(i+1)}{2} \right) = \frac{1}{2} \sum_{i=1}^{n-1} (i+1) = \frac{1}{2} \left(\frac{(n-1)(n+2)}{2} \right) = \Theta(n^2) \end{aligned}$$

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:

linked list. → array is not the preferred data structure for insertion sort

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.

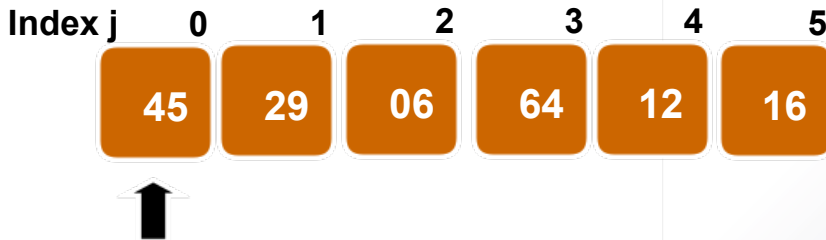
Summary

- 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.

Index j	0	1	2	3	4	5
	45	29	06	64	12	16

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.



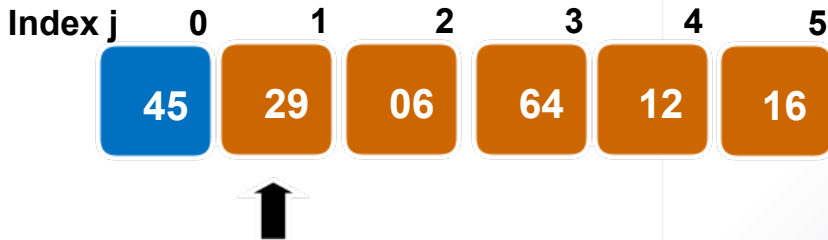
Summary

- 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.

Index j	0	1	2	3	4	5
	45	29	06	64	12	16

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:** $\mathcal{O}(n)$, when input array is already sorted.
 - **Worst case:** $\mathcal{O}(n^2)$, when input array is reversely sorted.
 - **Average case:** $\mathcal{O}(n^2)$.