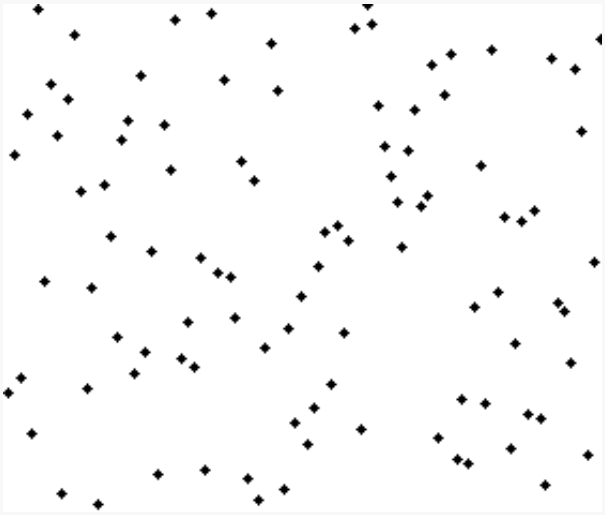
Data Structure & Algorithm

1. Sorting Algorithm

1.2 Insertion Sort

1.2.1 Brief

Insertion sort is a simple sorting algorithm, could be done in 3 lines of codes, which place one element on its correct order each time. The main idea of insertion sort is to build up the sorted array in-place, and insert the unsorted element into sorted array by inverse search and comparison.



(Image from Wikipedia)

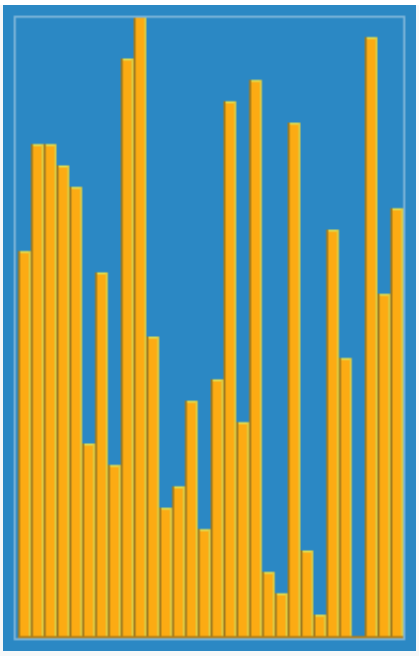
The average and worst time complexity of insertion sort is $O(n^2)$, the best case is $O(n)$ comparisons and $O(1)$ swaps. Compared with quicksort, insertion sort is stable but not efficient, it takes sequential swap operations while quicksort use idea of recursion. Compared with other quadratic sorting algorithm like selection sort and bubble sort, insertion sort is more efficient.

Insertion sort, as we mentioned in quicksort, performs well in small dataset so we could use insertion sort to optimize quicksort when the lenth of subarray is less than a threshold.

1.2.2 Algorithm

Suppose, we have an array of integers `A = [1,6,3,2,5,8,7,4]` as input and the supposed output after quicksort is `A= [1,2,3,4,5,6,7,8]`.

The main idea of insertion sort is to update the sorted array with one new element in unsorted part each iteration.



(Image from Wikipedia)

1. We choose the first number of array as sorted, the main iteration `i` will begin with the second element, which indicates the first element of unsorted array. Sub-iteration variable `j` is used for comparison, we will talk it soon.

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Pointer	j	i						
Value	**1**	6	3	2	5	8	7	4

2. Compare `A[i]` with sorted element one by one by sub-iteration `j` until find the correct position, indicating that is larger than left side elememnt and smaller than right side element. Then update the iteration variable, meaning that the elements before iteration variable are sorted.

   In this case, `A[j+1]=6>A[j]=1`, so 6 should put after 1, the sorted array is `[1,6]` now, iteration variables `i=i+1`, `j=i-1`.

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Pointer		j	i					
Value	**1**	**6**	3	2	5	8	7	4

   In this case, `A[j+1]=3<A[j]=6`, so 3 should put before 6. We swap `A[j]` with `A[j+1]` then `j=j-1`. Update sub-iteration variable `j` is to find somewhere to stop, meaning that find a element is less than the element we are going to sort.

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Pointer	j		i					
Value	**1**	3	**6**	2	5	8	7	4

   In this case, `A[j+1]=3>A[j]=1`, so 3 should put after 1. We find the correct position of 3. Then update the iteration

variables `i=i+1`, `j=i-1`.

Index	0	1	2	3	4	5	6	7
Pointer			j	i				
Value	**1**	**3**	**6**	2	5	8	7	4

Same as previous step, we find `A[j+1]=2<A[j]=6`, so swap 2 and 6, then `j=j-1`.

Index	0	1	2	3	4	5	6	7
Pointer		j		i				
Value	**1**	**3**	2	**6**	5	8	7	4

Still, `A[j+1]=2<A[j]=3`, swap and update `j`.

Index	0	1	2	3	4	5	6	7
Pointer	j			i				
Value	**1**	2	**3**	**6**	5	8	7	4

We find that `A[j+1]=2<A[j]=1`, so element 2 is in its correct position. Update iteration variables `i=i+1`, `j=i-1`.

Index	0	1	2	3	4	5	6	7
Pointer				j	i			
Value	**1**	**2**	**3**	**6**	5	8	7	4

Samely, we could replace the rest of array, unsorted array, in their right position.

Index	0	1	2	3	4	5	6	7
Pointer								
Value	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**

1.2.3 Python Code

Python

```python
A = [1,6,3,2,5,8,7,4]

def insert_sort(A):
    for i in range(1,len(A)):
        for j in range(i,0,-1):
            if A[j-1]>A[j]: A[j-1], A[j] = A[j], A[j-1]
    return A

insert_sort(A)
print A
```

1.2.4 C++ Code

```cpp
#include "iostream"
using namespace std;

void insert_sort(int A[],int lenth){
    for (int i=1;i<lenth;i++){
        for (int j=i;j>0;j--){
            if (A[j-1]>A[j]) swap(A[j-1],A[j]);
        }
    }
}

int main(){
    int A[]={1,6,3,2,5,8,7,4};
    int lenth = sizeof(A)/sizeof(A[0]);
    insert_sort(A, lenth);
    for (int i=0;i<lenth;i++){
        cout << A[i];
    }
    cout << endl;
    return 0;
}
```

C++

1.2.5 Time Complexity

Best case

The best case of insertion sort is sorted array. In this case, the whole algorithm process won't swap any elements and for each iteration only one comparison will happened, because sorted array must have `A[j+1]>A[j]`. So the time complexity is $O(n)$.

Worst case

Inversive sorted array is one of worst case. For all iterations, it needs `i` times compar3 and swap operations, and it easily to have the time complexity is $O(n^2)$.

Average case

In average case, we could imagine that the insertion sort need to swap half of elements in the array. So, we could say that the average case is also quadratic, which makes insertion sort impractical for sorting large arrays.