The evolution of insertion sort

Liu Xinyu *

January 5, 2013

1 Introduction

In previous chapter, we introduced the 'hello world' data structure, binary search tree. In this chapter, we explain insertion sort, which can be think of the 'hello world' sorting algorithm [1]. It's straightforward, but the performance is not as good as some divide and conqueror sorting approaches, such as quick sort and merge sort. Thus insertion sort is seldom used as generic sorting utility in modern software libraries. We'll analyze the problems why it is slow, and trying to improve it bit by bit till we reach the best bound of comparison based sorting algorithms, $O(N \lg N)$, by evolution to tree sort. And we finally show the connection between the 'hello world' data structure and 'hello world' sorting algorithm.

The idea of insertion sort can be vivid illustrated by a real life poker game[2]. Suppose the cards are shuffled, and a player starts taking card one by one.

At any time, all cards in player's hand are well sorted. When the player gets a new card, he insert it in proper position according to the order of points. Figure 1 shows this insertion example.

Based on this idea, the algorithm of insertion sort can be directly given as the following.

```
function SORT(A)
    X ← Φ
    for each x ∈ A do
        INSERT(X, x)
    return X
```

It's easy to express this process with folding, which we mentioned in the chapter of binary search tree.

$$insert = foldL \quad insert \quad \Phi \tag{1}$$

*Liu Xinyu
Email: liuxinyu95@gmail.com

[1] Some reader may argue that 'Bubble sort' is the easiest sort algorithm. Bubble sort isn't covered in this book as we don't think it's a valuable algorithm[1]

Figure 1: Insert card 8 to proper position in a deck.

Note that in above algorithm, we store the sorted result in X, so this isn't in-place sorting. It's easy to change it to in-place algorithm.

function SORT(A)
  for $i \leftarrow 2$ to LENGTH(A) **do**
    insert A_i to sorted sequence $\{A'_1, A'_2, ..., A'_{i-1}\}$

At any time, when we process the i-th element, all elements before i have already been sorted. we continuously insert the current elements until consume all the unsorted data. This idea is illustrated as in figure 2.

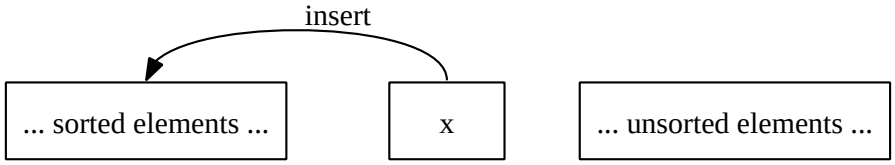

Figure 2: The left part is sorted data, continuously insert elements to sorted part.

We can find there is recursive concept in this definition. Thus it can be expressed as the following.

$$sort(A) = \begin{cases} \Phi & : \quad A = \Phi \\ insert(sort(\{A_2, A_3, ...\}), A_1) & : \quad otherwise \end{cases} \tag{2}$$

2 Insertion

We haven't answered the question about how to realize insertion however. It's a puzzle how does human locate the proper position so quickly.

For computer, it's an obvious option to perform a scan. We can either scan from left to right or vice versa. However, if the sequence is stored in plain array,

it's necessary to scan from right to left.

```
function SORT(A)
    for i ← 2 to LENGTH(A) do  ▷ Insert A[i] to sorted sequence A[1...i − 1]
        x ← A[i]
        j ← i − 1
        while j > 0 ∧ x < A[j]  do
            A[j + 1] ← A[j]
            j ← j − 1
        A[j + 1] ← x
```

One may think scan from left to right is natural. However, it isn't as effect as above algorithm for plain array. The reason is that, it's expensive to insert an element in arbitrary position in an array. As array stores elements continuously. If we want to insert new element x in position i, we must shift all elements after i, including $i+1, i+2, ...$ one position to right. After that the cell at position i is empty, and we can put x in it. This is illustrated in figure 3.

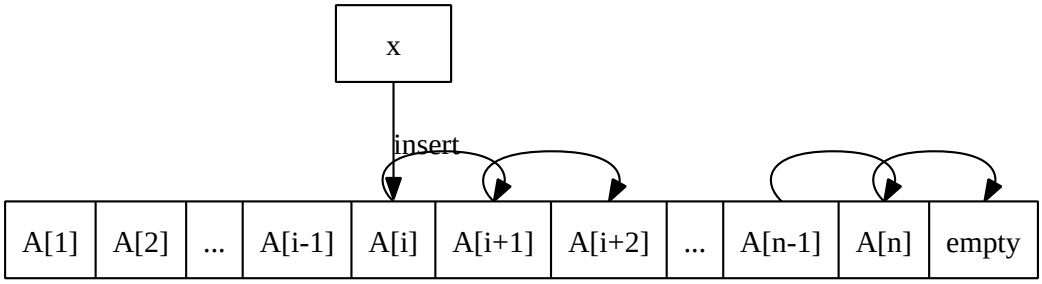

Figure 3: Insert x to array A at position i.

If the length of array is N, this indicates we need examine the first i elements, then perform $N - i + 1$ moves, and then insert x to the i-th cell. So insertion from left to right need traverse the whole array anyway. While if we scan from right to left, we totally examine the last $j = N - i + 1$ elements, and perform the same amount of moves. If j is small (e.g. less than $N/2$), there is possibility to perform less operations than scan from left to right.

Translate the above algorithm to Python yields the following code.

```
def isort(xs):
    n = len(xs)
    for i in range(1, n):
        x = xs[i]
        j = i - 1
        while j >= 0 and x < xs[j]:
            xs[j+1] = xs[j]
            j = j - 1
        xs[j+1] = x
```

It can be found some other equivalent programs, for instance the following ANSI C program. However this version isn't as effective as the pseudo code.

```
void isort(Key* xs, int n){
  int i, j;
  for(i=1; i<n; ++i)
    for(j=i-1; j>=0 && xs[j+1] < xs[j]; --j)
      swap(xs, j, j+1);
}
```

This is because the swapping function, which can exchange two elements typically uses a temporary variable like the following:

```
void swap(Key* xs, int i, int j){
  Key temp = xs[i];
  xs[i] = xs[j];
  xs[j] = temp;
}
```

So the ANSI C program presented above takes $3M$ times assignment, where M is the number of inner loops. While the pseudo code as well as the Python program use shift operation instead of swapping. There are $N + 2$ times assignment.

We can also provide INSERT() function explicitly, and call it from the general insertion sort algorithm in previous section. We skip the detailed realization here and left it as an exercise.

All the insertion algorithms are bound to $O(N)$, where N is the length of the sequence. No matter what difference among them, such as scan from left or from right. Thus the over all performance for insertion sort is quadratic as $O(N^2)$.

Exercise 1

- Provide explicit insertion function, and call it with general insertion sort algorithm. Please realize it in both procedural way and functional way.

3 Improvement 1

Let's go back to the question, that why human being can find the proper position for insertion so quickly. We have shown a solution based on scan. Note the fact that at any time, all cards at hands have been well sorted, another possible solution is to use binary search to find that location.

We'll explain the search algorithms in other dedicated chapter. Binary search is just briefly introduced for illustration purpose here.

The algorithm will be changed to call a binary search procedure.

function SORT(A)

```
for i ← 2 to LENGTH(A) do
    x ← A[i]
    p ← BINARY-SEARCH(A[1...i − 1], x)
    for j ← i down to p do
        A[j] ← A[j − 1]
    A[p] ← x
```

Instead of scan elements one by one, binary search utilize the information that all elements in slice of array $\{A_1, ..., A_{i-1}\}$ are sorted. Let's assume the order is monotonic increase order. To find a position j that satisfies $A_{j-1} \leq x \leq A_j$. We can first examine the middle element, for example, $A_{\lfloor i/2 \rfloor}$. If x is less than it, we need next recursively perform binary search in the first half of the sequence; otherwise, we only need search in last half.

Every time, we halve the elements to be examined, this search process runs $O(\lg N)$ time to locate the insertion position.

```
function BINARY-SEARCH(A, x)
    l ← 1
    u ← 1+ LENGTH(A)
    while l < u do
        m ← ⌊(l+u)/2⌋
        if A_m = x then
            return m                          ▷ Find a duplicated element
        else if A_m < x then
            l ← m + 1
        else
            u ← m
    return l
```

The improved insertion sort algorithm is still bound to $O(N^2)$, compare to previous section, which we use $O(N^2)$ times comparison and $O(N^2)$ moves, with binary search, we just use $O(N \lg N)$ times comparison and $O(N^2)$ moves.

The Python program regarding to this algorithm is given below.

```
def isort(xs):
    n = len(xs)
    for i in range(1, n):
        x = xs[i]
        p = binary_search(xs[:i], x)
        for j in range(i, p, -1):
            xs[j] = xs[j-1]
        xs[p] = x

def binary_search(xs, x):
    l = 0
    u = len(xs)
    while l < u:
        m = (l+u)/2
```

```
        if xs[m] == x:
            return m
        elif xs[m] < x:
            l = m + 1
        else:
            u = m
    return l
```

Exercise 2

Write the binary search in recursive manner. You needn't use purely functional programming language.

4 Improvement 2

Although we improve the search time to $O(N \lg N)$ in previous section, the number of moves is still $O(N^2)$. The reason of why movement takes so long time, is because the sequence is stored in plain array. The nature of array is continuously layout data structure, so the insertion operation is expensive. This hints us that we can use linked-list setting to represent the sequence. It can improve the insertion operation from $O(N)$ to constant time $O(1)$.

$$insert(A, x) = \begin{cases} \{x\} & : & A = \Phi \\ \{x\} \cup A & : & x < A_1 \\ \{A_1\} \cup insert(\{A_2, A_3, ...A_n\}, x) & : & otherwise \end{cases} \quad (3)$$

Translating the algorithm to Haskell yields the below program.

```
insert :: (Ord a) ⇒ [a] → a → [a]
insert [] x = [x]
insert (y:ys) x = if x < y then x:y:ys else y:insert ys x
```

And we can complete the two versions of insertion sort program based on the first two equations in this chapter.

```
isort [] = []
isort (x:xs) = insert (isort xs) x
```

Or we can represent the recursion with folding.

```
isort = foldl insert []
```

Linked-list setting solution can also be described imperatively. Suppose function KEY(x), returns the value of element stored in node x, and NEXT(x) accesses the next node in the linked-list.

function INSERT(L, x)
  $p \leftarrow NIL$
  $H \leftarrow L$
  while $L \neq NIL \wedge$ KEY(L) $<$ KEY(x) **do**

$p \leftarrow L$
$L \leftarrow \textsc{Next}(L)$
$\textsc{Next}(x) \leftarrow L$
if $p \neq NIL$ **then**
$H \leftarrow x$
else
$\textsc{Next}(p) \leftarrow x$
return H

For example in ANSI C, the linked-list can be defined as the following.

```
struct node{
  Key key;
  struct node* next;
};
```

Thus the insert function can be given as below.

```
struct node* insert(struct node* lst, struct node* x){
  struct node *p, *head;
  p = NULL;
  for(head = lst; lst && x→key > lst→key; lst = lst→next)
    p = lst;
  x→next = lst;
  if(!p)
    return x;
  p→next = x;
  return head;
}
```

Instead of using explicit linked-list such as by pointer or reference based structure. Linked-list can also be realized by another index array. For any array element A_i, $Next_i$ stores the index of next element follows A_i. It means A_{Next_i} is the next element after A_i.

The insertion algorithm based on this solution is given like below.

function $\textsc{Insert}(A, Next, i)$
$j \leftarrow \perp$
while $Next_j \neq NIL \wedge A_{Next_j} < A_i$ **do**
$j \leftarrow Next_j$
$Next_i \leftarrow Next_j$
$Next_j \leftarrow i$

Here $\perp$ means the head of the $Next$ table. And the relative Python program for this algorithm is given as the following.

```
def isort(xs):
    n = len(xs)
    next = [-1]*(n+1)
    for i in range(n):
        insert(xs, next, i)
    return next
```

```
def insert(xs, next, i):
    j = -1
    while next[j] != -1 and xs[next[j]] < xs[i]:
        j = next[j]
    next[j], next[i] = i, next[j]
```

Although we change the insertion operation to constant time by using linked-list. However, we have to traverse the linked-list to find the position, which results $O(N^2)$ times comparison. This is because linked-list, unlike array, doesn't support random access. It means we can't use binary search with linked-list setting.

Exercise 3

- Complete the insertion sort by using linked-list insertion function in your favorate imperative programming language.
- The index based linked-list return the sequence of rearranged index as result. Write a program to re-order the original array of elements from this result.

5 Final improvement by binary search tree

It seems that we drive into a corner. We must improve both the comparison and the insertion at the same time, or we will end up with $O(N^2)$ performance.

We must use binary search, this is the only way to improve the comparison time to $O(\lg N)$. On the other hand, we must change the data structure, because we can't achieve constant time insertion at a position with plain array.

This remind us about our 'hello world' data structure, binary search tree. It naturally support binary search from its definition. At the same time, We can insert a new leaf in binary search tree in $O(1)$ constant time if we already find the location.

So the algorithm changes to this.

function SORT(A)
  $T \leftarrow \Phi$
  for each $x \in A$ **do**
    $T \leftarrow$ INSERT-TREE(T, x)
  return TO-LIST(T)

Where INSERT-TREE() and TO-LIST() are described in previous chapter about binary search tree.

As we have analyzed for binary search tree, the performance of tree sort is bound to $O(N \lg N)$, which is the lower limit of comparison based sort[3].

6 Short summary

In this chapter, we present the evolution process of insertion sort. Insertion sort is well explained in most textbooks as the first sorting algorithm. It has simple and straightforward idea, but the performance is quadratic. Some textbooks stop here, but we want to show that there exist ways to improve it by different point of view. We first try to save the comparison time by using binary search, and then try to save the insertion operation by changing the data structure to linked-list. Finally, we combine these two ideas and evolute insertion sort to tree sort.

References

[1] http://en.wikipedia.org/wiki/Bubble_sort

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. "Introduction to Algorithms, Second Edition". ISBN:0262032937. The MIT Press. 2001

[3] Donald E. Knuth. "The Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)". Addison-Wesley Professional; 2 edition (May 4, 1998) ISBN-10: 0201896850 ISBN-13: 978-0201896855