Insertion Sort

- a sorting technique that incrementally builds a sorted list by inserting a new element in the correct place in an already sorted (sub-)list.

- It starts by trivially including the 1st element of the unsorted array into an initially empty sublist (or subarray).

e.g.: consider \(100, 50, 75, 6, 125, 11, 90\).

Insertion sort proceeds as follows:

iteration 1: "sorted" subarray = \(100\)

iteration 2: " " = \(50, 100\)

(created by inserting \(50\) before \(100\))

iter 3: "sorted subarray" = \(50, 75, 100\)

(Inserting \(75\) before \(100\))

iter 4: 6 50 75 100

iter 5: 6 50 75 100 125

iter 6: 6 11 50 75 100 125

iter 7: 6 11 50 75 90 100 125.

The important point to note is that you insert a new element \(x\) in the position right before the smallest element that is greater than \(x\) (or right after the largest element smaller than \(x\)).

Although we refer to "list" or "sub-list" in this discussion, these are used in a
figurative sense. Insertion sort can also be implemented using an array.

Implementation using array (Program 2.14 of Sahni - Chapter 2)

```c
void insertion_sort (int *arr, int n)
{
    for (i = 1; i < n; i++)
    {
        temp = arr[i];
        insert (arr, i, temp);
    }
}
```

```c
void insert (int *arr, int n, int x)
{
    for (i = n-1; i > 0; i--)
    {
        if (x >= arr[i]) break;
        arr[i+1] = arr[i];
    }
    arr[i+1] = x; // "shift elements greater than x to the right"
    "i+1" is the correct index to insert x.
"if x >= arr[i], then i+1 is the correct place for x."
```
Dry run

100  50  75  6  125  11  90  arr

arr[0] = 100

\[ i = 1 \text{  (in the routine "insert-sort")} \]

\[ \text{insert} (arr, 1, 50) \rightarrow 100 150 75 6 125 11 90 \]
\[ a[i+1] = x \to 50 100 75 6 125 11 90 \]

\[ i = 2 \rightarrow \text{insert} (arr, 2, 75) \rightarrow 50 100 150 6 125 11 90 \]
\[ 50 75 100 6 125 11 90 \]

\[ i = 3 \rightarrow \text{insert} (arr, 3, 6) \rightarrow 50 50 75 100 125 11 90 \]
\[ 6 50 75 100 125 11 90 \]

\[ i = 4 \rightarrow \text{insert} (arr, 4, 125) \rightarrow 6 50 75 100 125 11 90 \]

\[ i = 5 \rightarrow \text{insert} (arr, 5, 11) \rightarrow 6 50 50 75 100 125 11 \]
\[ 6 11 50 75 100 125 90 \]

\[ i = 6 \rightarrow \text{insert} (arr, 6, 90) \rightarrow 11 50 75 100 125 \]
\[ 6 11 50 75 90 100 125 \]

Time Complexity:

The outer for loop runs for \( n-1 \) iterations.

In the best case (if the array were already sorted), each insert call will take constant time. So the best case complexity is \( O(n) \).

The worst case (if you had to traverse down to the beginning of the array each time insert is called) complexity is \( (n-1)n = O(n^2) \).

\[
\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} = O(n^2)
\]
Algorithm development: Data structures to manipulate polynomials.

A polynomial of degree \( n \) is a function of the form
\[
p(x) = \sum_{i=0}^{n} a_i x^i.
\]
It can be represented in a computer in the form of an array or a linked list.

**Array Based Representation**

A polynomial of degree \( n \) → float array \( \mathbf{A} \) of size \( n \) → \( \mathbf{A}[i] = \text{coefficient of the term } x^i \text{ in } p(x) \). (The coefficient could be 0).

Consider addition of two polynomials:

\[
p_a(x) = \sum_{i=0}^{n_a} a_i x^i \quad \text{and} \quad p_b(x) = \sum_{i=0}^{n_b} b_i x^i
\]

having degree \( n_a \) and \( n_b \) respectively,

\[
p_a(x) + p_b(x) = \sum_{i=0}^{\min(n_a,n_b)} (a_i + b_i) x^i + \sum_{i=\min(n_a,n_b)+1}^{\max(n_a,n_b)} c_i x^i
\]

where \( c_i = b_i \) (if \( n_b > n_a \)), \( \forall i, \quad n_a+1 \leq i \leq n_b \)

\[
= a_i \quad \text{(if } n_b < n_a \text{),} \quad \forall i, \quad n_b+1 \leq i \leq n_a
\]
```c
float * addpoly (float *a, float *b, int na, int nb)
{
    float *c = new float [max(na, nb)];
    for (i = 0; i < min(na, nb); i++)
    {
        c[i] = a[i] + b[i];
    }
    if (na > nb)
    {
        for (i = min(na, nb) to max(na, nb) - 1)
        {
            c[i] = a[i];
        }
    }
    else if (nb > na)
    {
        for (i = nb; i < na; i++)
        {
            c[i] = b[i];
        }
    }
    return c;
}
```
Linked list representation

The disadvantage of the array-based polynomial representation is that a lot of memory is wasted on zero-valued coefficients which occur in polynomials of the form
\[ p(x) = x^{10} + 1 \] (say).

In the linked list representation, we maintain a node for each term of the polynomial. The node contains the degree of the term as well as the coefficient. By convention, we maintain the list sorted by degree (in ascending order) of the term.
So the representation for \[ p(x) = x^{10} + 3x^7 + 2x^5 + 10x + 11 \]
looks like:

```
11, 0 -> 10, 1 -> 2, 5 -> 3, 7 -> 1, 10
```

We will now write code to add two polynomials represented as degree-sorted linked lists:

```c
float *add_poly_lists (float *a, float *b, int na, int nb)
{
    node = node

    float *c = new node;
    c->coeff = INVALID; // header node of
    c->degree = INVALID; // the new list
    c->next = NULL;

    node *savedc = c;
```
\[ a = a \rightarrow \text{next} \quad b = b \rightarrow \text{next} \]

while ( \( a \neq \text{NULL} \) && \( b \neq \text{NULL} \))

\{
  \text{if} (a \rightarrow \text{degree} < b \rightarrow \text{degree})
  \{
    c \rightarrow \text{next} = \text{new node};
    c \rightarrow \text{degree} = a \rightarrow \text{degree} ;
    c \rightarrow \text{coeff} = a \rightarrow \text{coeff} ;
    c = c \rightarrow \text{next} ;
    a = a \rightarrow \text{next} ;
  \}

  \text{elseif} (a \rightarrow \text{degree} > b \rightarrow \text{degree})
  \{
    c \rightarrow \text{next} = \text{new node} ;
    c \rightarrow \text{degree} = b \rightarrow \text{degree} ;
    c \rightarrow \text{coeff} = a \rightarrow \text{coeff} ;
    c = c \rightarrow \text{next} ;
    b = b \rightarrow \text{next} ;
  \}

  \text{elseif} (a \rightarrow \text{degree} == b \rightarrow \text{degree})
  \{
    c \rightarrow \text{next} = \text{new node} ;
    c \rightarrow \text{degree} = a \rightarrow \text{degree} ;
    c \rightarrow \text{coeff} = a \rightarrow \text{coeff} + b \rightarrow \text{coeff} ;
    c = c \rightarrow \text{next} ;
    a = a \rightarrow \text{next} ;
    b = b \rightarrow \text{next} ;
  \}
\}

PTO
if (a == NULL && b != NULL) {
    while (b != NULL) {
        c = new node;
        c = degree = degree;
        c = coeff = coeff;
        c = c = next;
        b = b = next;
    }
}
else if (b == NULL && a != NULL) {
    while (a != NULL) {
        c = new node;
        c = degree = degree;
        c = coeff = coeff;
        c = c = next;
        a = a = next;
    }
}
return c;