

Chapter 30 – Introduction to Sorting

1. Why Sort?

Unordered lists of information have several drawbacks. When an unordered list is searched, the searching algorithm may have to check every element in the list to determine that the desired item is not to be found. If a large enough unordered list is searched frequently, this can lead to a significant lost time, as the resulting search of an unordered list is considered to be $O(n^2)$ in the worst case.

A list that is ordered (a sorted list) can be searched in much less time than an unordered list, even if the desired item is not among the elements of the list. In fact, an ordered list can be searched in $O(\log_2 n)$ time, which is substantially better than $O(n)$. If an ordered list is searched a great many times, it can be done in $O(n \log_2 n)$, which is far superior to a worst case search time when repeatedly searching an unordered list of $O(n^2)$.

2. An N^2 Sort: Selection Sort

The first sort that is usually learned by a 3rd generation language programmer falls into a category of algorithms called brute force sorts. The advantage of brute force sorts is their simplicity, usually requiring some form of nested counting loops and an *if* statement. The disadvantage of brute force sorts is that they are not efficient in time. If data in a list is volatile, that is it changes a lot, the list would have to be resorted frequently and a brute force sort would be a serious disadvantage. If the contents of the list are relatively stable, a brute force sort may not be too much of a disadvantage.

We will start by learning the *selection sort*, which takes an existing list and sorts its elements into ascending order (smallest to largest) or descending order (largest to smallest). To do this, the selection sort algorithm compares every element in the list against every other element in the list. Each time the values of the two elements being compared are out of the desired order, the values are swapped. This is done in a systematic way through a nesting of 2 loops, usually 2 *for-loops*. The index for the outer loop is used to mark the place of one element. The index of the inner loop is used to find all of the other elements that will be compared against the element given by the index of the outer loop. This can be written as follows:

```

for (long index1=0; index1 < max-1; index1++)
    for (long index2 = index1+1; index2 < max; index2++)
        if (array[index1] > array[index2]) {
            temp = array[index1];
            array[index1] = array[index2];
            array[index2] = temp;
        }

```

This example sorts the values in array into ascending order. (See the Boolean expression in the if statement before the swap of element values.)

In brief, this selection sort of a list into ascending order works by repeatedly searching for the minimum value in *array* and placing it at the location in the array indicated by the value of *index1*. As *index1* increases in value, any value in *array* that has an index less than *index1* is already sorted. The for-loop of *index2* performs the search for the minimum value located at indices greater than the value of *index1*. As *index2* increases in value, the values at *array[index2]* are compared against the value at *array[index1]*. If the value at *array[index1]* is greater than the value at *array[index2]*, the values are swapped. A third variable (*temp*) of the same type or class of the list element is required to exchange two values. When the *for-index2* loop terminates, the value at *array[index1]* is guaranteed to be the smallest value from *array[index1]* to *array[max-1]*. Since the value of *index2* always begins at one greater than *index1*, only the remaining unsorted values in the array are searched.

The following is an example of how selection sort works.

```
a[0]=5      index1
a[1]=15     index2
a[2]=6
a[3]=9
a[4]=3
```

```
a[0]=5      index1
a[1]=15
a[2]=6      index2
a[3]=9
a[4]=3
```

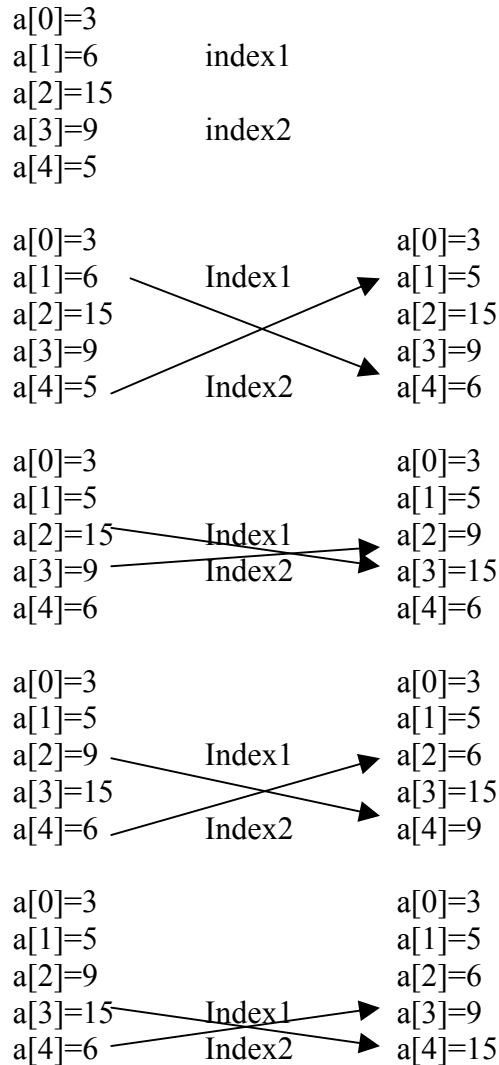
```
a[0]=5      index1
a[1]=15
a[2]=6
a[3]=9      index2
a[4]=3
```

```
a[0]=5      index1
a[1]=15
a[2]=6
a[3]=9
a[4]=3      index2
```

```
a[0]=3
a[1]=15
a[2]=6
a[3]=9
a[4]=5
```

```
a[0]=3
a[1]=15     index1
a[2]=6      index2
a[3]=9
a[4]=5
```

```
a[0]=3
a[1]=6
a[2]=15
a[3]=9
a[4]=5
```



The selection sort algorithm consists of two for-loops that depend on the number of elements in the list (n) for its growth in time. The most frequently executed statement in a nest of two for-loops is the second for statement. The outer for statement is executed n times. The inner for statement is executed on average $\frac{1}{2}n$ times and is started $n-1$ times. This means that the inner for statement is executed $\frac{1}{2}n(n-1)$, which is $\frac{1}{2}n^2 - \frac{1}{2}n$. This gives a worst case growth in time to the selection sort of the curve described by $O(n^2)$. (Note: The data structure *array* is also dependent on the number of elements in the list, but exists before the algorithm, will exist after the algorithm and its size is not effected by the algorithm.)

Exercises

1. What is the output of the following code?

```
a = 3;
b = 4;
a = b;
cout <<a << " " << b;
```

2. What is the output of the following code?

```
a = 3;
b = 4;
t = b;
b = a;
a = t;
cout <<a << " " << b << " " << t;
```

3. What is the output of the following code?

```
a[0] = 5; a[1] = 2; a[2]=4; a[3]=3;
for (int x=0; x<4; x++) {
    for (int y=0; y<4; y++)
        if (a[x] > a[y]) {
            t = a[x];
            a[x] = a[y];
            a[y] = t;
        }
    cout << a[x] << endl;
}
```

Programming Assignment 30.1

Using the following function header, create a function that sorts a list in an array consisting of elements of type integer into ascending order.

```
void sort(int a[ ]; long listlength)
```

Programming Assignment 30.2

Using the function of 30.1, sort and output a list of random integers. Determine the actual time taken to sort the list (the time after the list is created and before it is output) when the list is 1,000 integers, 10,000 integers, 100,000 integers and 1,000,000 integers.

3. An N^2 Sort: Insertion Sort

In-order insertion of an element into a sorted list returns a sorted list that is one element longer. An empty list is by definition sorted. The insertion sort algorithm starts with an empty list and inserts one element, in-order, at a time into the list. The result after each insertion is a sorted list.

The following is an outline of the insertion algorithm (item is the value to be inserted). It will insert an element into a list that is sorted in ascending order. The resulting larger list will also be sorted in ascending order.

```

index = listlength;
while (index > 0 && array[index-1] > item) {
    array[index] = array[index-1];
    index--;
}
array[index] = item;
listlength++;

```

The in order insertion algorithm works by starting at the last index and working toward the 0 index. Each element is copied from its index to its index+1. When the correct index is found for the item to be inserted, the item is simply copied into that element.

To make an insertion sort, the insertion algorithm is called many times, once for each element to be inserted.

The insertion algorithm scans on average $\frac{1}{2}$ the list before inserting the new element. The while statement is the most frequently executed statement, on average $\frac{1}{2}n$ times. If there are n elements to be inserted into the list, the insertion algorithm must be called n times. This gives a worst case growth curve in time of $\frac{1}{2}n^2$, or as expressed in Omicron notation, $O(n^2)$. Since the insertion sort directly effects the amount of memory required based on n , it has a growth curve in space of $O(n)$.

Programming Assignment 30.3

Using the following function header, create a function that inserts an element into a list in ascending order in an array consisting of elements of type integer. The variable object *item* contains the new element to be inserted.

```
void insert(int a[ ]; long & listlength, int item)
```

Programming Assignment 30.4

Using the function of 30.3, sort and output a list of random integers. Using a stopwatch, determine the time taken to sort the list (the time after the list is created and before it is output) when the list is 1,000 integers, 10,000 integers, 100,000 integers and 1,000,000 integers. How do these times compare with the selection sort?

Programming Assignment 30.5

Change the program of 30.4 so that the array is dynamic. Begin with the size of 100 elements and expand the array by 100 elements each time the limit is reached. Using a stopwatch, determine the time taken to sort the list (the time after the list is created and before it is output) when the list is 1,000 integers, 10,000 integers, 100,000 integers and 1,000,000 integers. How do these times compare with the selection sort and the insertion sort of 30.4? What has changed in the analysis of the growth in time of the insertion sort algorithm?