Non-Recursive Sorting Algorithms

Topics:

- ○ General Sorting Algorithm Structure
- ○ Bubble Sort
- ○ Insertion Sort
- ○ Selection Sort

Resources:

- `Array.h`
- `Repository.h`
- `main.cpp`

Introduction

Sorting is the process of organizing comparable objects. When a list of objects is sorted, we can write more efficient algorithms to search or modify the list. The sorting process, however, is not unique. There are several ways to sort. In this lecture, we will look at three non-recursive sorting methods, namely, bubble sort, insertion sort and selection sort.

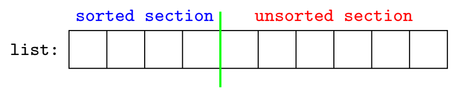
General Sorting Algorithm Structure

In general, when a sequential sorting algorithm does not create additional list to accomplish its goal, it must implement the `swap()` algorithm. This algorithm is a simple function that swaps the values of two objects. Its definition is as follows

```
template <typename T>
void Swap(T& a,T& b)
{
 T t = a;
 a = b;
 b = t;
}
```

It creates an object to store one of the values of the two objects. Next, it assigns the object, which has its value stored in the new object, the value of the other object. And last, it assigns the other object to value stored in the new object. A very simple, but important process.

Furthermore, these sorting algorithms divides the list into two sections we will call the *sorted section* and *unsorted section* by a pivot (or index)



The objective of these algorithms is to increase the sorted section while decreasing the unsorted section until the unsorted section is empty. This process requires traversing through the list as well as adding to the sorted section. Hence, the algorithm has two essential components. The algorithm that is responsible for adding an element to the sorted section we will call the *core*; and, the traversal algorithm we will call the *main*. The main algorithm is a linear traversal that either states from the beginning of the list and goes to the end of the list element by element, or vica versa. Coding this algorithm should already be well known. The core algorithm will be nested in the main algorithm and it is unique for each of the following sorting algorithms.

Bubble Sort

For each instance of the core algorithm of the bubble sort, it compares, and when necessary, swaps adjacent elements of the unsorted section of the list starting from the beginning until the end of the unsorted section. At the end of the run of the core, the maximum value in the unsorted section will be in the last element of the unsorted section, which is now the new beginning of the sorted section. Its code is as follows

```
template <typename T>
void BCore(Array<T>& data,int n)
{
 for(int i = 1;i <= n;i += 1)
 {
  if(data[i-1] > data[i])
  {
   Swap(data[i-1],data[i]);
  }
 }
}
```

where n is the index of the last element in the unsorted section. Let us calculate the runtime of this function. Assume that all operations have a processing cost of 1.

cost	**time**	**operation**
1	1	`int i = 1`
1	$n+1$	`i <= n`
1	n	`data[i-1] > data[i]`
1	n	`Swap(data[i-1],data[i]);`
1	n	`i += 1`

$t(n) = 4n + 2$

Now, let us complete the definition for the bubble sort algorithm and calculate its runtime function. To accurately sort the list, the bubble sort must traverse the list starting from the last element and ending with the first element. Hence, the code for the bubble sort is

```
template <typename T>
void BubbleSort(Array<T>& data)
{
 for(int i = data.Size()-1;i > 0;i -= 1)
 {
  BCore(data,i);
 }
}
```

And its runtime function is

cost	**time**	**operation**
1	1	`int i = data.Size()-1`
1	n	`i > 0`
1	$t(i)$	`BCore(data,i);`
1	$n-1$	`i -= 1`
$T(n) = 2n^2 + 2n - 2$		

where n refers to the size of the list, and since

$$\begin{aligned}
\sum_{i=1}^{n-1} t(i) &= \sum_{i=1}^{n-1} 4i + 2 \\
&= \sum_{i=1}^{n-1} 4i + \sum_{i=1}^{n-1} 2 \\
&= 4\sum_{i=1}^{n-1} i + 2\sum_{i=1}^{n-1} 1 \\
&= 4\left(\frac{(n-1)(n)}{2}\right) + 2(n-1) \\
&= 2(n-1)(n) + 2(n-1) \\
&= 2(n-1)(n+1) \\
&= 2(n^2-1) \\
&= 2n^2 - 2
\end{aligned}$$

which means

$$\begin{aligned}
T(n) &= 1 + n + 2n^2 - 2 + n - 1 \\
&= 2n^2 + 2n - 2
\end{aligned}$$

Thus, the Big-O runtime of the bubble sort function is $O(n^2)$ (quadratic). Last the bubble sort written without the function call of `BCore()` is

```
template <typename T>
void BubbleSort(Array<T>& data)
{
 for(int i = data.Size()-1;i > 0;i -= 1)
 {
  for(int j = 1;j <= i;j += 1)
  {
   if(data[j-1] > data[j])
   {
    Swap(data[j-1],data[j]);
   }
  }
 }
}
```

Insertion Sort

For each instance of the core algorithm of the insertion sort, it compares and swaps adjacent elements of the sorted section of the list starting from the end, which is the beginning of the unsorted section, until either the value is in the correct location of the sorted section or it reaches the beginning of the sorted section. At the end of the run of the core, the new value will be in its sorted position. Its code is as follows

```
template <typename T>
void ICore(Array<T>& data,int n)
{
 int i = n;

 while(i > 0 && data[i] < data[i-1])
 {
  Swap(data[i],data[i-1]);
  i -= 1;
 }
}
```

where n is the index of the first element in the unsorted section. Let us calculate the runtime of this function. Assume that all operations have a processing cost of 1.

cost	time	operation
1	1	`int i = n;`
1	$n+1$	`i > 0 && data[i] < data[i-1]`
1	n	`Swap(data[i],data[i-1]);`
1	n	`i -= 1;`

$t(n) = 3n + 2$

Now, let us complete the definition for the insertion sort algorithm and calculate its runtime function. To accurately sort the list, the insertion sort must traverse the list starting from second element in the list and ending with the last element. Hence, the code for the insertion sort is

```
template <typename T>
void InsertionSort(Array<T>& data)
{
 for(int i = 1;i < data.Size();i += 1)
 {
  ICore(data,i);
 }
}
```

And its runtime function is

cost	time	operation
1	1	`int i = 1`
1	n	`i < data.Size()`
1	$t(i)$	`ICore(data,i);`
1	$n-1$	`i += 1`

$T(n) = \frac{3}{2}n^2 + \frac{5}{2}n - 2$

where n refers to the size of the list, and since

$$\begin{aligned}
\sum_{i=1}^{n-1} t(i) &= \sum_{i=1}^{n-1} 3i+2 \\
&= \sum_{i=1}^{n-1} 3i + \sum_{i=1}^{n-1} 2 \\
&= 3\sum_{i=1}^{n-1} i + 2\sum_{i=1}^{n-1} 1 \\
&= 3\left(\frac{(n-1)(n)}{2}\right) + 2(n-1) \\
&= \frac{3(n-1)(n)}{2} + \frac{4(n-1)}{2} \\
&= \frac{(n-1)(3n+4)}{2} \\
&= \frac{3n^2+n-4}{2} \\
&= \frac{3}{2}n^2 + \frac{1}{2}n - 2
\end{aligned}$$

which means

$$\begin{aligned}
T(n) &= 1 + n + \frac{3}{2}n^2 + \frac{1}{2}n - 2 + n - 1 \\
&= \frac{3}{2}n^2 + \frac{5}{2}n - 2
\end{aligned}$$

Thus, the Big-O runtime of the insertion sort function is O(n^2) (quadratic). Last the insertion sort written without the function call of `ICore()` is

```
template <typename T>
void InsertionSort(Array<T>& data)
{
 for(int i = 1;i < data.Size();i += 1)
 {
  int j = i;

  while(j > 0 && data[j] < data[j-1])
  {
   Swap(data[j],data[j-1]);
   j -= 1;
  }
 }
}
```

Selection Sort

For each instance of the core algorithm of the selection sort, it finds the index of the minimum value in the unsorted section of the list starting from the beginning, which is the end of the sorted section, until the end of the unsorted section. Afterwards, it swaps the values of the first element and the element with the minimum value if they are different. At the end of the run of the core, the new value will be added to the end of the sorted section. Its code is as follows

```
template <typename T>
void SCore(Array<T>& data,int i)
{
 int m = i;

 for(int j = i + 1;j < data.Size();j += 1)
 {
  if(data[m] > data[j])
  {
   m = j;
  }
 }

 if(i != m)
 {
  Swap(data[i],data[m]);
 }
}
```

where i is the index of the first element in the unsorted section. Let us calculate the runtime of this function. Assume that all operations have a processing cost of 1.

cost	**time**	**operation**
1	1	`int m = i;`
1	1	`int j = i + 1;`
1	r	`j < data.Size()`
1	$r-1$	`data[m] > data[j]`
1	$r-1$	`m = j;`
1	$r-1$	`j += 1`
1	1	`i != m`
1	1	`Swap(data[i],data[m]);`

$t(r) = 4r + 1$

where $r = n - i$ and n is the size of the list.

Now, let us complete the definition for the selection sort algorithm and calculate its runtime function. To accurately sort the list, the selection sort must traverse the list starting from the first element in the list and ending with the last element. Hence, the code for the selection sort is

```
template <typename T>
void SelectionSort(Array<T>& data)
{
 for(int i = 0;i < data.Size();i += 1)
 {
  SCore(data,i);
 }
};
```

And its runtime function is

cost	**time**	**operation**
1	1	`int i = 0`
1	$n+1$	`i < data.Length()`
1	$t(i)$	`SCore(data,i);`
1	n	`i += 1`

$T(n) = 2n^2 + 2n + 2$

where n refers to the size of the list, and since

$$\begin{aligned}
\sum_{i=1}^{n} t(i) &= \sum_{i=1}^{n} t(n-i) \\
&= \sum_{i=1}^{n} 4(n-i)+1 \\
&= \sum_{i=1}^{n} 4n - \sum_{i=1}^{n} 4i + \sum_{i=1}^{n} 1 \\
&= 4n\sum_{i=1}^{n} 1 - 4\sum_{i=1}^{n} i + 2\sum_{i=1}^{n} 1 \\
&= 4n(n) - 4\left(\frac{(n)(n+1)}{2}\right) + 2n \\
&= 4n^2 - 2(n^2+n) + 2n \\
&= 4n^2 - 2n^2 - 2n + 2n \\
&= 2n^2
\end{aligned}$$

which means

$$\begin{aligned}
T(n) &= 1 + n + 1 + 2n^2 + n \\
&= 2n^2 + 2n + 2
\end{aligned}$$

Thus, the Big-O runtime of the insertion sort function is O(n^2) (quadratic). Last the insertion sort written without the function call of `SCore()` is

```
template <typename T>
void SelectionSort(Array<T>& data)
{
 for(int i = 0;i < data.Size();i += 1)
 {
  int m = i;

  for(int j = i + 1;j < data.Size();j += 1)
  {
   if(data[m] > data[j])
   {
    m = j;
   }
  }

  if(i != m)
  {
   Swap(data[i],data[m]);
  }
 }
};
```