7SENG010W Data Structres & Algorithms

Week 2 Lecture

Arrays

Overview of Week 2 Lecture: Arrays

Aim is to introduce the array data structure & several of the most common array searching & sorting algorithms:

- *Array Data Structures*
  - Properties of Arrays
  - Examples of Arrays: 1D, 2D, 3D, jagged
- *Searching Algorithm*
  - Binary Search
- *Sorting Algorithms*
  - Selection sort
  - Bubble sort
  - Merge sort
- *.NET Array Collection Class*
  - ArrayList

PART I

Array Data Structures

Properties of Array Data Structures

Arrays are one of the simplest *collection* data structures:

- *Linear*: collection of data organised using a *sequence* structure.
- *Static*: standard arrays have a *fixed size*, but most languages also support more versatile *dynamic* arrays that allow elements to be added or deleted.
- *Indexed*:
  - Data items accessed via one or more *indexes* into the arrays, e.g. `numbers[5]`, `matrix[5,6]`.
  - C♯ array *indexes* start at 0 & run up to the length of the array minus 1.
  - For an array of 32 bit integer (4 bytes) each array element takes up 4 bytes of memory.
  - So assuming that `array[0]` is stored in *memory location* l_0 then `array[1]` is stored in *memory location* $l_1 = l_0 + 4$, etc.
- *Array Algorithms*: usually involve `for`-loops or recursion, & a collection of array *indexes* as *meta data* used to keep track of the *current state of the algorithm* as it processes the array.
- Examples: Arrays, Matrices (2-dimensional Arrays), n-dimensional Arrays, etc.

C♯ System.Array Class

C♯ supports single-dimensional arrays, multidimensional arrays, & *jagged arrays* (arrays of different sized arrays).

```
// 1-dimensional array of ints, Objects
int[] myIntArray = new int[5] { 1, 2, 3, 4, 5 } ;

myIntArray[2] = 33 ;

// 2-dimensional array of Doubles,
Double[,] myDoubles = new Double[10, 20];

myDoubles[2,5] = 2.022 ;

// 3-dimensional array of Strings
String[,,] myStrings = new String[5, 3, 10] ;

myStrings[4,2,8] = "Dog" ;

// jagged array: 3 rows of 3 different sized arrays
int[][] jaggedInts = new int[3][] ;

jaggedInts[0] = new int[2] ;  // 1st row: 2 element array
jaggedInts[1] = new int[4] ;  // 2nd row: 4 element array
jaggedInts[2] = new int[6] ;  // 3rd row: 6 element array

jaggedInts[2][5] = 99 ;       // last row & last element
```

PART II

Array Searching Algorithm: Binary Search

Searching Algorithm: Binary Search

- We saw a very simple *Linear Search* algorithm in the previous lecture, it is a *Brute Force* $O(N)$ algorithm, that simply used a `for`-loop to search each element in an array for a specific value.

- *Binary Search* is a (better) *Divide & Conquer* algorithm, because if it does not find the value it repeatedly divides the "*search space*" in half.

- To apply the *Binary Search* to a container it must have two properties:

- *Indexable:* the algorithm defines a "*segment*" of the container (e.g. array) to search & the middle value in the segment to check.

  Array *segment* is defined using 2 array indexes, e.g. for a 10 element array: **start** = 0, **end** = 9 & **middle** = (**start** + **end**)/2 = (0+9)/2 = 4.

- *Sorted:* the container's elements must be *sorted*, e.g. ascending order.

  This allows the algorithm to decide which part of the array to search next if the search value is not equal to the **middle** value then:

less:	search the lower array segment, i.e. left of **middle**
greater:	search the upper array segment, i.e. right of **middle**

Binary Search: Pseudo code

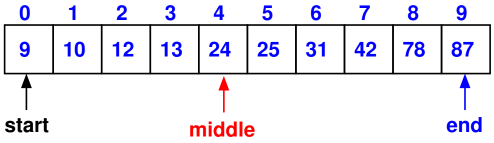
```
BinarySearch( ARRAY values, VALUE valueToFind )
BEGIN
   // define array segment to search, initially the whole array
   start <-- start of array segment
   end   <-- end of array segment

   WHILE ( non empty array segment )
   BEGIN
        // so search the array segment, check middle element
        middle <-- middle of array segment: start .. end

        IF ( valueToFind == values[middle] )
             RETURN middle
        ELSE
            IF( valueToFind < values[middle] )
                // search lower segment (left of middle)
                start <-- start          // same start
                end   <-- middle - 1     // new end
            ELSE
                IF ( values[middle] < valueToFind )
                   // search upper segment (right of middle)
                   start <-- middle + 1     // new start
                   end   <-- end            // same end
                ENDIF
            ENDIF
        ENDIF
   ENDWHILE

   RETURN NOT_FOUND ;  // value Not found (or FALSE )
END
```

Example Binary Search (steps 1 & 2)

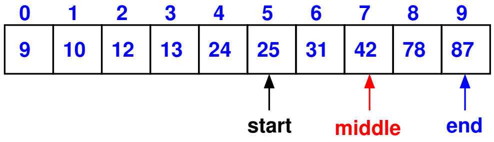
Search for **31** in the following sorted array of numbers:

0	1	2	3	4	5	6	7	8	9
9	10	12	13	24	25	31	42	78	87

- **Step 1:** set array indexes as **start** = 0, **end** = 9 & **middle** = (0+9)/2 = 4; 31 is greater than 24, so search *upper segment*.

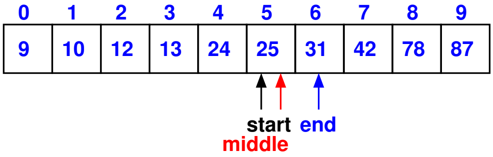


- **Step 2:** set *upper segment* indexes: **start** = 5, **end** = 9 & **middle** = (5+9)/2 = 7; 31 is less than 42, so search *lower segment*.

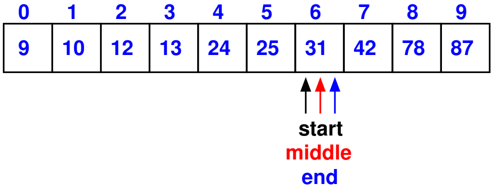


Example Binary Search (steps 3 & 4)

- **Step 3:** set *lower segment* indexes: **start** = 5, **end** = 6 & **middle** = (5+6)/2 = 5; 31 is greater than 25, so look at *upper segment*.



- **Step 4:** set *upper segment* indexes: **start** = 6, **end** = 6 & **middle** = (6+6)/2 = 6; 31 is equal to 31, so *found 31*.



Example Binary Search: Failed Search

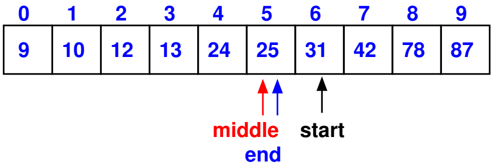
If in the previous search we had instead been searching for any of:

$25 < \textbf{26, 27, 28, 29, 30} < 31$

then none of the above numbers would have been found in step 4.

Then since all of the above are less than 31, the algorithm would then decide to search in the *lower segment*.

- **Step 5:** set *lower segment* indexes: **start** = 6, **end** = 5 & **middle** = (6+5)/2 = 5;



But now **start** $\leq$ **end** is false, i.e. **start** $>$ **end**, so this is an **empty segment**.

Therefore, there is no further segment to search, the search terminates & hence the *search would fail* to find any of 26, 27, 28, 29, 30, as expected.

Analysis of Binary Search Algorithm

- Each iteration cuts the search segment (roughly) in half.
- If the array is size N, and N is less than 2^K for some number K:
  - After 1 iteration there are less than 2^{K-1} candidates left
  - After 2 iterations there are less than 2^{K-2} candidates left
  - $\dots$
  - After $K-1$ iterations there are less than 2 candidates left
- The search finishes after at most $K = \lceil \log_2(N) \rceil$ iterations
- So complexity of *Binary Search* is $O(\log_2(N))$.
- **Example:** $N = 10$, & $2^3 < 10 < 2^4$ which is $8 < 10 < 16$, so $K = 4$.

  After 1 iteration there are less than $2^{K-1} = 2^3 = 8$ candidates left
  After 2 iterations there are less than $2^{K-2} = 2^2 = 4$ candidates left
  After 3 iterations there are less than $2^{K-3} = 2^1 = 2$ candidates left
  After 4 iterations there are less than $2^{K-4} = 2^0 = 1$ candidate left

  So for $n = 10$, the search finished after at most $4 = \lceil \log_2(10) \rceil$ iterations.

PART III

Array Sorting Algorithms:

Selection Sort, Bubble Sort & Merge Sort

Categories of Sorting Algorithms

Many sorting algorithms fall into two categories:

- *Divide & Conquer Algorithms:*
  - basic approach is to split the work (e.g. process input data) into a number (usually 2) smaller sub-problems & process the sub-problems separately.
  - May need to combine the solved sub-problems for a complete solution.
  - Recursion is often used to solve sub-problems.
  - Examples: Binary Search, Merge Sort.
- *Decrease & Conquer Algorithms:*
  - basic approach is that on each "*iteration*" of the algorithm, a number of "*units of work*" are done that *decreases* the remaining amount of work required.
  - The number of "*units of work*" is usually: a *constant*, e.g. 1; a *constant factor*, e.g. 2, halving the work; *variable factor*, varies for each iteration.
  - Examples: Selection Search, (Bubble Sort?).

Note that many basic algorithms are often not categorised & are just designated as "*a simple algorithm*", e.g. Selection Search, Bubble Sort.

The Sorting problem

We will be dealing with the following problem:

- **Input:** a sequential data structure
  - We will be working with arrays
  - The data will be integers for simplicity
- For example:

```
// unsorted array:
int[] unsortedNumbers = { 91, 32, 92, 13, 73, 14 } ;
```

- **Goal:** produce a *permutation* of the contents such that they are in *ascending* (or *descending*) order.
- For example:

```
// sorted array:
int[] sortedNumbers = { 13, 14, 32, 73, 91, 92 } ;
```

The Sorting problem Questions & Issues

The main questions & issues are

- What algorithms exist?

  Quite a few; we will only look at a small sample

- What is their time complexity?

- What is the best we can hope for?

- For the last two, we first define our *atomic operations*, so when we analyse the algorithms to determine their order of complexity what actions do we need to consider:
  - Pairwise comparison: `if ( a[i] < a[j] ) { ... }`
  - Swap: `swap(array, i, j)` exchanges the two values at `array[i]` & `array[j]`
  - Usually every swap will be preceded by at least one comparison, but not every comparison will be followed by a swap.
  - So the number of comparisons will be the dominating factor.

Sorting Algorithm 1: Selection Sort

Selection Sort is a simple sorting algorithm.

- In Selection Sort the array consists of
  - An **unsorted segment** at the beginning of the array.
  - Initially the whole array is the **unsorted segment**.
  - A **sorted segment** at the end of the array.
  - Initially the **sorted segment** is empty.
- We shrink the **unsorted** segment & grow the **sorted segment** in each iteration using these steps:
  1. Find the **maximal element** of the **unsorted** segment.
  2. Move it to the end of the **unsorted** segment; by swapping it with the **element** at the end of the **unsorted** segment
  3. This swapped **maximal element** in effect then becomes part of the **sorted** segment, i.e. is moved from the **unsorted** segment to the **sorted** segment.
  4. Repeat until the **unsorted** segment is empty & hence the **sorted** segment is the whole array of sorted elements.

Example of Selection Sort

We begin with our example array:

```
// unsorted array:
int[] numbers = { 91, 32, 92, 13, 73, 14 } ;
```

Run *Selection Sort* algorithm:

- **Step 0:** The **unsorted** segment is the whole array & the **sorted** segment is empty.
  Initialise the *last unsorted index* (lusInd), the *max value* (maxVal) & its *index* (maxInd) in the **unsorted** segment.

  91, 32, 92, 13, 73, 14 [lusInd = 5, maxInd = 0, maxVal = 91]

- **Step 1:** Find the **largest** value **92** (using sequential search) & the **last** value in the **unsorted** segment **14** & swap them:

  91, 32, 92, 13, 73, 14 [lusInd = **5**, maxInd = **2**, maxVal = **92**]

  91, 32, 14, 13, 73, 92

Example Selection Sort (continued)

- **Step 2:** Find the **largest** value **91** & the **last** value in the **unsorted** segment **73** & swap them:

  91, **32, 14, 13**, **73**, **92** [lusInd = **4**, maxInd = **0**, maxVal = **91**]

  73, **32, 14, 13**, **91, 92**

- **Step 3:** Find the **largest** value **73** & the **last** value in the **unsorted** segment **13** & swap them:

  73, **32, 14**, **13**, **91, 92** [lusInd = **3**, maxInd = **0**, maxVal = **73**]

  13, **32, 14**, **73, 91, 92**

- **Step 4:** Find the **largest** value **32** & the **last** value in the **unsorted** segment **14** & swap them:

  13, **32**, **14**, **73, 91, 92** [lusInd = **2**, maxInd = **1**, maxVal = **32**]

  13, **14**, **32, 73, 91, 92**

Example of Selection Sort (continued)

- **Step 5:** Find the **largest** value **14** & the **last** value in the **unsorted** segment which is also **14** & “swap” them:

  13, **14**, **32, 73, 91, 92** [lusInd = **1**, maxInd = **1**, maxVal = **14**]

  13, **14, 32, 73, 91, 92**

- **Step 6:** Find the **largest** value **13** & the **last** value in the **unsorted** segment which is also **13** & “swap” them:

  13, **14, 32, 73, 91, 92** [lusInd = **0**, maxInd = **0**, maxVal = **13**]

  13, 14, 32, 73, 91, 92

- **Termination:** the **unsorted** segment is empty & all values are in the **sorted** segment.

Selection Sort: implementation

```
public class SelectionSort
{
   public static void sort(int[] values)
   {
      int lastUnsorted = values.length - 1 ; // end of the unsorted segment

      while ( lastUnsorted > 0 )
      {
         // find the maximal unsorted element
         int maxIndex = 0 ;              // its index
         int maxValue = values[0] ;      // its value

         for ( int i = 1 ; i <= lastUnsorted ; i++ )
         {
            if ( values[i] > maxValue )
            {
                // new maximal value at i
                maxIndex = i ;
                maxValue = values[i] ;
            }
         }

         // swap maximal with last unsorted, & add it to the sorted segment
         values[maxIndex]     = values[lastUnsorted] ;
         values[lastUnsorted] = maxValue ;
         lastUnsorted-- ;
      }
   }
}
```

Selection Sort: analysis

- How many operations does this algorithm perform on an array of size N?

- There are N *iterations of the main loop*, in each iteration:
  1. Searches for the **largest element** in the **unsorted** segment.
  2. A *single swap*, adding it to the **sorted** segment.

- If K is the current size of the **unsorted** segment, then it takes $K-1$ comparisons to find the **largest element**.

- The **unsorted** section *shrinks by 1 element each iteration*, so:
  - *1st iteration:* **unsorted** segment is $K = N$, requires $K-1 = N-1$ comparisons

    2nd iteration: its K−1, requires $(K-1)-1 = N-2$ comparisons.

    ...
  - *Total comparisons* is: $(N-1)+(N-2)+...+2+1$
    Using standard formula for sum of $1..(N-1)$ this gives $(N^2-N)/2$ total comparisons.

- Hence the running time for Selection sort is: $T(N) = (N^2-N)/2$

- *Big-O* is only concerned with the *dominant term*, so it ignores constant multiplying factor $1/2$ & the $-N$, so this gives $O(T(N)) = O(N^2)$.

Sorting Algorithm 2: Bubble Sort

- Bubble Sort is similar to Selection Sort.
- While going through the **unsorted** segment, the **largest value** found so far "*bubbles*" up.
- This means that it is moved towards the **sorted** segment at the end of the array, similar to Selection Sort.
- But the difference being that, any element `array[j]` found to be greater than its right neighbour `array[j+1]` is swapped with it.
- So there may be *many swaps per iteration* through the array, not just the one as in Selection Sort.
- The overall effect is that usually more elements are moved around the array on each iteration.
- But after each *outer iteration* the **largest value** found in the **unsorted** segment is *bubbled* towards the **sorted** segment of the array, & eventually added to it.

Bubble Sort Example

Using the previous example array:

91, 32, 92, 13, 73, 14

Initially the **unsorted** segment is the whole array, the **sorted** segment is empty.

Focus on the "**bubble**" element & its **comparison** element.

- *1st outer iteration* — **6 unsorted**, **0 sorted**

  Inner iterations:

  1. **91**, **32**, **92, 13, 73, 14** — **91** $>$ **32** so swap them.
  2. **32**, **91**, **92**, **13, 73, 14** — **91** $<$ **92** do not swap, bubble set to **92**.
  3. **32, 91**, **92**, **13**, **73, 14** — **92** $>$ **13** so swap them.
  4. **32, 91, 13**, **92**, **73**, **14** — **92** $>$ **73** so swap them.
  5. **32, 91, 13, 73**, **92**, **14** — **92** $>$ **14** so swap them.
  6. **32, 91, 13, 73, 14**, **92** — no more comparisons, end of inner iterations.

So **92** has been "bubbled up", i.e. moved to the **sorted** segment.

Bubble Sort Example (continued)

- *2nd outer iteration* — **5 unsorted**, **1 sorted**

  Inner iterations:

  1. **32**, **91**, **13, 73, 14, 92** — **32** $<$ **91**, bubble set to **91**
  2. **32**, **91**, **13**, **73, 14**, **92**
  3. **32, 13**, **91**, **73**, **14**, **92**
  4. **32, 13, 73**, **91**, **14**, **92**
  5. **32, 13, 73, 14**, **91, 92** – **91** added to **sorted** segment.

- *3rd outer iteration* — **4 unsorted**, **2 sorted**

  Inner iterations:

  1. **32**, **13**, **73, 14**, **91, 92**
  2. **13**, **32**, **73**, **14**, **91, 92** — **32** $<$ **73**, bubble set to **73**
  3. **13, 32**, **73**, **14**, **91, 92**
  4. **13, 32, 14**, **73, 91, 92** – **73** added to the **sorted** segment.

Bubble Sort Example (continued)

- *4th outer iteration* — **3 unsorted**, **3 sorted**

  Inner iterations:

  1. **13**, **32**, **14**, **73, 91, 92** — **13** $<$ **32**, bubble set to **32**
  2. **13**, **32**, **14**, **73, 91, 92**
  3. **13, 14**, **32, 73, 91, 92** – **32** added to the **sorted** segment.

- *5th outer iteration* — **2 unsorted**, **4 sorted**

  Inner iterations:

  1. **13**, **14**, **32, 73, 91, 92** — **13** $<$ **14**, bubble set to **14**
  2. **13**, **14, 32, 73, 91, 92** – **14** added to the **sorted** segment.

- *6th outer iteration* — **1 unsorted**, **5 sorted**

  Inner iterations:

  1. **13, 14, 32, 73, 91, 92** – 13 added to the **sorted** segment.

Algorithm terminated: **0 unsorted**, **6 sorted**.

Analysis of Bubble Sort Algorithm

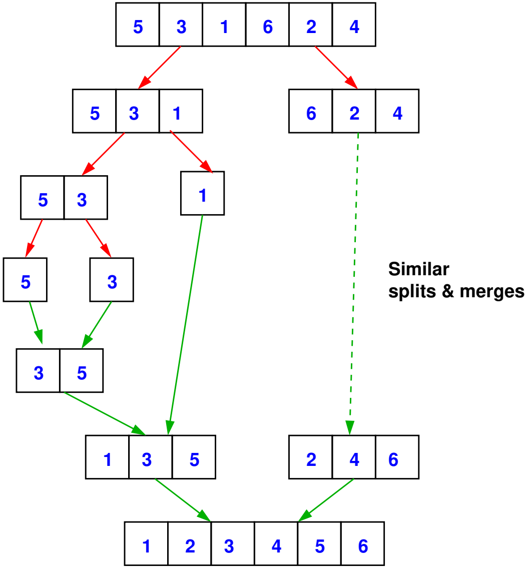
- Implementation of Bubble Sort is left as a tutorial exercise.
- The complexity analysis looks just like Selection Sort
- On each iteration of the main outer loop, for an **unsorted** segment of size K elements then:
    - Shrinks the **unsorted** segment by 1, i.e. results in an **unsorted** segment of size $K - 1$.
    - Requires $K - 1$ comparisons, to compare the K **unsorted** elements.
    - Requires *up to* $K - 1$ *swaps*, the actual number depends on how many elements are less than the **bubble** element.
- So the order of complexity (for the worst case) is again $O(N^2)$.
- Q: Can we do better?
  A: *Yes*, see the next sorting algorithm.

Sorting Algorithm 3: Merge Sort

- Suppose we want to apply the *Divide & Conquer* algorithm strategy to the sorting problem.
- Recall that the *Divide & Conquer* algorithm strategy entails:
  - Dividing a problem into several sub-problems of the same type & similar size.
  - Solve sub-problems using recursion.
  - If necessary, sub-problems solutions are *combined* for a solution to the original problem.
- A straightforward translation of the Divide & Conquer approach into steps is:
  - If there are 0 or 1 values, there is **no sorting to do**, so finished.
  - Otherwise, if there are 2 or more values:
    1. *Split* the array into (roughly) two equal halves. (See *Binary Search* algorithm.)
    2. *Recursively sort* each half.
    3. *Combine* the sorted halves by *merging them*.
- These steps are an overview of the *Merge Sort* algorithm.

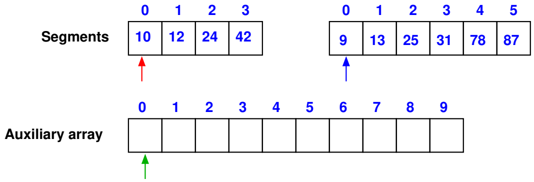
Overview of Merge Sort Algorithm

The *Merge Sort* algorithm applied to an array of 6 numbers.



Merge Sort: example of the merging step

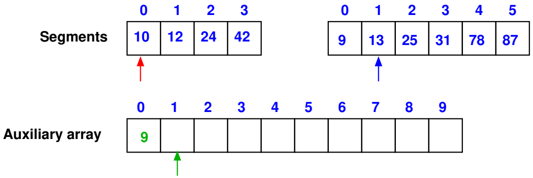
- In the final steps of Merge Sort, we need to *merge two sorted parts into one*, so consider the following *merging* example.
- Initial state of the 2 segments & an auxiliary array used for merging:



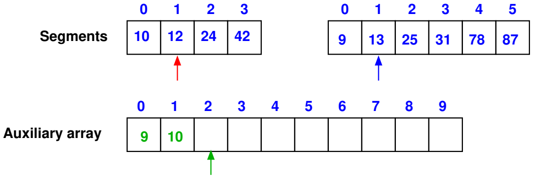
- Merging 2 segments:
  1. Start at the beginning of both segments, by using an index to point to the first element in the segment.
  2. As long as there are *unused elements in **both** segments*:
     2.1 Using the two indexes, compare the two elements they point to.
     2.2 Copy the smaller one into the temporary array, increase the index.
     2.3 In copied from segment, increase the index to point to the next element.
  3. Copy the remaining unmerged elements into the array.
  4. Then copy the sorted values back to the main array

Merge Sort Example: the merging step (continued)

- **Step 1:** compare the two pointed to values in the segments **10** & **9**; copy **9** into the auxiliary array; move segment's pointer to next element; move auxiliary's pointer to next free slot.

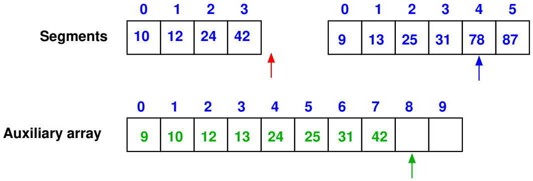


- **Step 2:** compare the pointed to values **10** & **13**; copy **10** into the auxiliary array, adjust pointers.

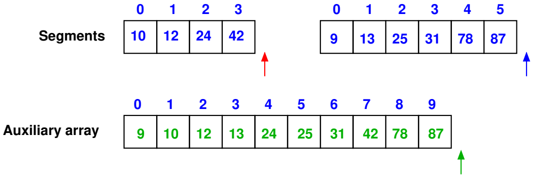


Merge Sort Example: the merging step (continued)

- **Step 8:** After six more merges all the elements in the left segment have been added to the auxiliary array, leaving **78** & **87** in the right segment.



- **Step 9:** add the right segment's remaining unmerged values **78** & **87** to the auxiliary array.



- **Termination:** all values from the two segments have been merged, so the merging sub-algorithm terminates.

Merge Sort: implementation

```
public class MergeSort
{
  // Merge sorted segments: values[first]..values[mid] &
  //                        values[mid+1]..values[last]
  private static void mergeRanges( int[] values, int first, int mid, int last )
  {
    Console.WriteLine(" Tutorial Exercise: TO DO ") ;
  }

  // Sorts the range: values[first]..values[last]
  private static void sortRange( int[] values, int first, int last )
  {
     if ( last > first )
     {
       // Otherwise there is nothing to do (single value)
       int mid = (first + last) / 2;          // split array into 2 segments

       sortRange( values, first, mid ) ;      // Recursively sort lower segment
       sortRange( values, mid + 1, last ) ;   // Recursively sort upper segment

       mergeRanges( values, first, mid, last ) ;   // Merge sorted segments
     }
  }

  public static void sort( int[] numbers )
  {
    sortRange( numbers, 0, numbers.Length - 1 ) ;
  }
}
```

Analysis of the Merge Sort Algorithm

Finding the complexity of recursive algorithms is usually quite hard.

But for many Divide & Conquer algorithms it can be done using the *Masters Theorem*.

So for the case of using Merge Sort on an array segment of size N, we get:

- The cost of merging two array segments of total size N is $O(N)$.
- Suppose $T(N)$ is the cost of using Merge Sort then for:
- *Trivial case $N = 1$:* $T(1) = 0$
- *Non-Trivial case $N > 1$:* we create 2 (roughly equal sized) sub-ranges of size $N/2$, sort them, & then merge them.

  So in this case $T(N) = T(N/2) + T(N/2) + O(N) \; = \; 2T(N/2) + O(N)$
- By the *Master Theorem*, in this case $T(N)$ is $O(N \log_2(N))$.
- Therefore, the order of complexity of Merge Sort is $O(N \log_2(N))$.

Note that $O(N \log_2(N))$ is the *best* **worst case** order of complexity that can be achieved for any sorting algorithm of N items.

PART IV

C♯/.NET Array Classes

C♯ Array Class Algorithms

- The `System.Array` class also includes multiple versions of methods for *Sorting* an array (17 versions) & performing a *Binary Search* on an array (8 versions).

- Examples of the basic versions of the sorting & searching methods:

```
// search entire array for value
public static int BinarySearch( Array array, object? value ) ;

// Search in range index .. index+(length-1) of array for value
public static int BinarySearch( Array array, int index, int length,
                                object? value ) ;


// Sorts the elements in an entire array
public static void Sort( Array array ) ;

// Sorts elements in range index .. index+(length-1)
public static void Sort( Array array, int index, int length ) ;
```

- See more details & example programs at Microsoft .NET documentation for the `BinarySearch` and `Sort(Array)` methods.

ArrayList Class

In the `System.Collections` namespace there is the `ArrayList` class.

This is an array whose size can be *dynamically* increased or decreased as required.

Example:

```
ArrayList myAL = new ArrayList() ;

myAL.Add("Hello") ;  // Add 3 strings
myAL.Add("World") ;
myAL.Add("!") ;

myAL.RemoveAt( 1 ) ;        // Removes element at index 1, "World"

myAL.Insert( 1, "Earth" ) ; // Inserts "Earth" at index 1

System.Console.WriteLine( myAL[0] ) ;  // prints "Hello"
System.Console.WriteLine( myAL[1] ) ;  // prints "Earth"
System.Console.WriteLine( myAL[2] ) ;  // prints "!"
```