Divide and Conquer Algorithms Merge Sort Algorithm

In this week

- Revising Recursion and Recursive Algorithms
- Revising Complexity of Algorithms
- Introduction to Divide and Conquer Algorithms
- Merge Sort Algorithm

Recursion and Recursive Algorithms

- **Definition:-** Recursion is the process by which the solution of a problem is given in terms of the solution of a sub problem of the same problem and where
  - There is a base case in which case the solution is given explicitly, and
  - The sub problem is closer to the base case than the original problem
- C++ allows recursion by enabling us to call a function from within the function itself

Recursion and Recursive Algorithms

- In C++ a function is recursive if the function calls itself
- A recursive C++ function may or may not return a value
- Recursion is a powerful programming technique
- However it has limitations: we can call a function for a limited number of times before the memory of the computer is all used
- Many problems that are hard to solve without recursion are easily solved using recursion

Recursion and Recursive Algorithms

- For example, consider the factorial problem defined for non negative integers as follows

$$n! = \begin{cases} n*(n-1)! & if \quad n > 0 \\ 1 & if \quad n = 0 \end{cases}$$

- We would like to write a C++ function that implements the function the way it is defined
- We see that the factorial of n is defined in terms of the factorial of (n-1)
- Thus it is recursive and a recursive C++ function is needed to implement it the way it is defined
- See below

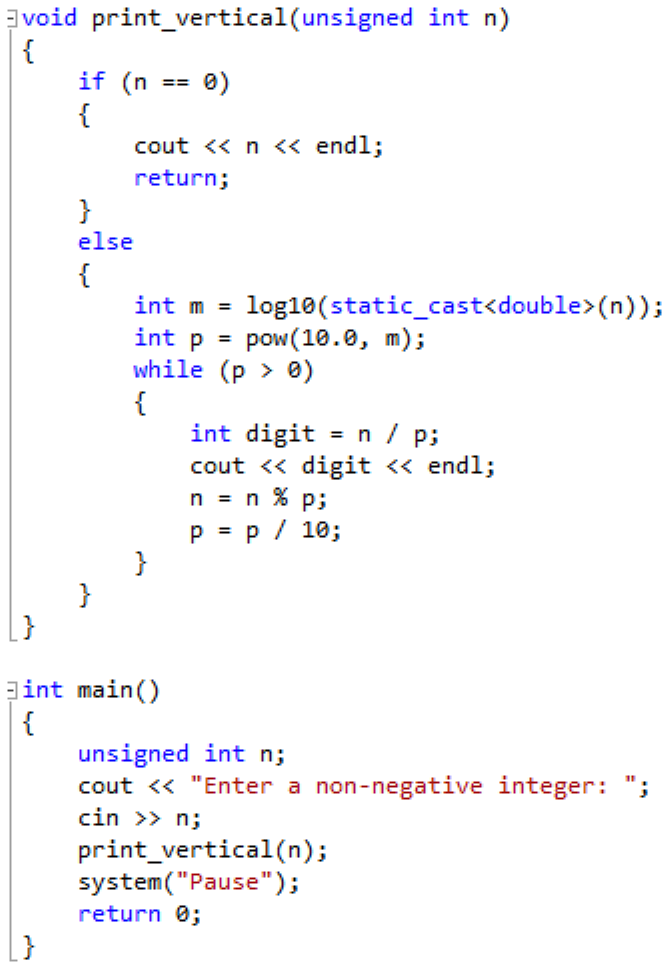
Recursion and Recursive Algorithms

- For example consider the following problem:
  - **Printing a Number Vertically:-** Write a C++ function named print_vertical that takes a non-negative integer argument and prints the non-negative integer argument vertically. For example, if the argument is 7249 then the function must print

    ```
    7
    2
    4
    9
    ```

Recursion and Recursive Algorithms

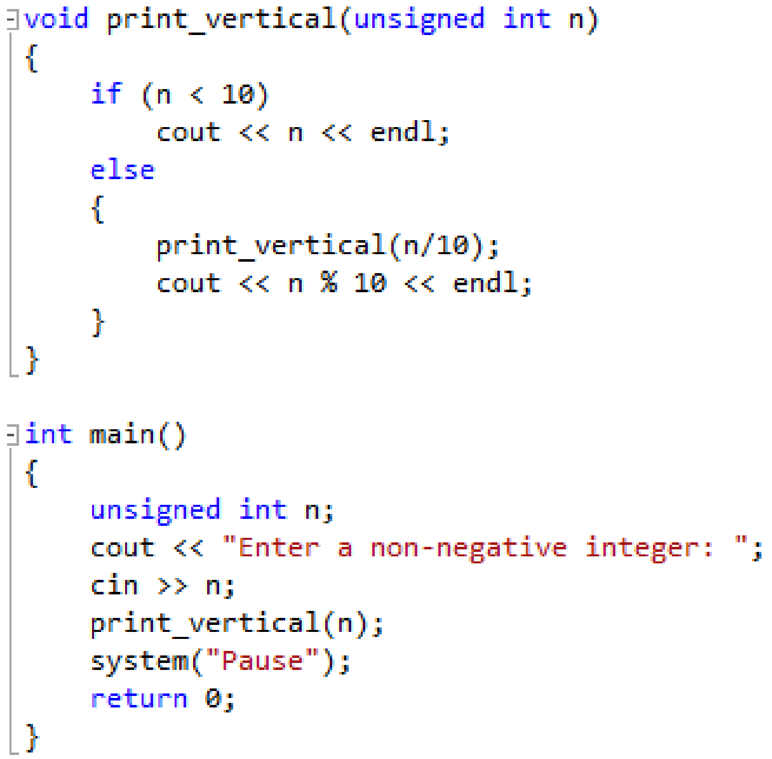
- A non-recursive function to accomplish this task might look like:



```
void print_vertical(unsigned int n)
{
    if (n == 0)
    {
        cout << n << endl;
        return;
    }
    else
    {
        int m = log10(static_cast<double>(n));
        int p = pow(10.0, m);
        while (p > 0)
        {
            int digit = n / p;
            cout << digit << endl;
            n = n % p;
            p = p / 10;
        }
    }
}

int main()
{
    unsigned int n;
    cout << "Enter a non-negative integer: ";
    cin >> n;
    print_vertical(n);
    system("Pause");
    return 0;
}
```

Recursion and Recursive Algorithms

- A more natural answer might be a recursive solution as follows:



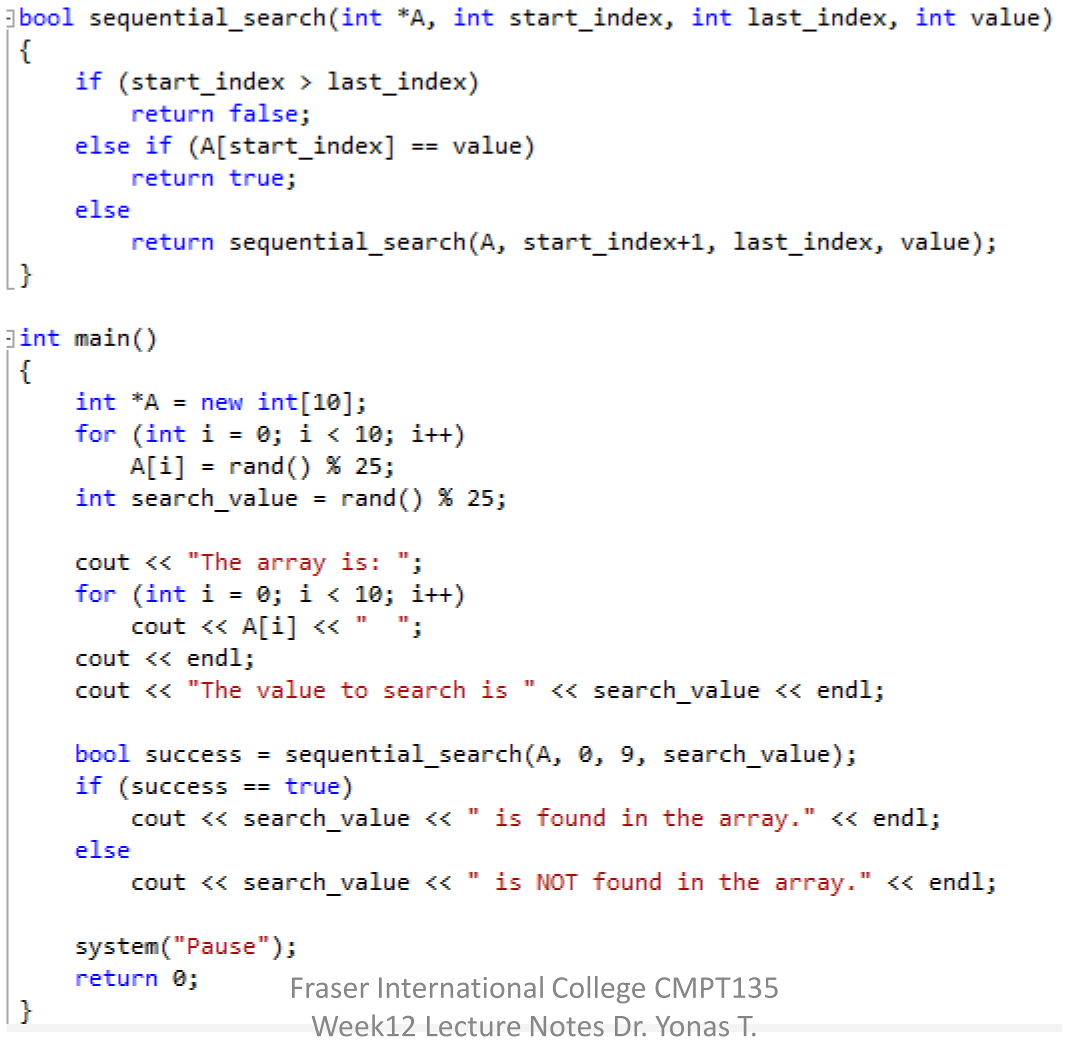
```
void print_vertical(unsigned int n)
{
    if (n < 10)
        cout << n << endl;
    else
    {
        print_vertical(n/10);
        cout << n % 10 << endl;
    }
}

int main()
{
    unsigned int n;
    cout << "Enter a non-negative integer: ";
    cin >> n;
    print_vertical(n);
    system("Pause");
    return 0;
}
```

Recursion and Recursive Algorithms

- More examples of problems that are elegantly solved with recursion:
  - **Sequential Search**: In order to search for a specific value in an array, we may do as follows:
    - If the array is empty then return NOT FOUND
    - Else if the first element of the array is equal to the search value then return FOUND
    - Else search the sub array starting from the next element in the array to the last element
  - Here the base case is when the array is empty

Recursion and Recursive Algorithms

- A recursive solution is therefore



```
bool sequential_search(int *A, int start_index, int last_index, int value)
{
    if (start_index > last_index)
        return false;
    else if (A[start_index] == value)
        return true;
    else
        return sequential_search(A, start_index+1, last_index, value);
}

int main()
{
    int *A = new int[10];
    for (int i = 0; i < 10; i++)
        A[i] = rand() % 25;
    int search_value = rand() % 25;

    cout << "The array is: ";
    for (int i = 0; i < 10; i++)
        cout << A[i] << "  ";
    cout << endl;
    cout << "The value to search is " << search_value << endl;

    bool success = sequential_search(A, 0, 9, search_value);
    if (success == true)
        cout << search_value << " is found in the array." << endl;
    else
        cout << search_value << " is NOT found in the array." << endl;

    system("Pause");
    return 0;
}
```

Recursion and Recursive Algorithms

- It is possible a recursive C++ function may call itself more than once. For example
  - **Towers of Hanoi Problem:-** Given three pegs named **A**, **B** and **C** and **n** number of discs stacked in peg **A** (increasing size top to bottom), we are required to move the **n** discs from peg **A** to peg **C** using peg **B** as temporary holder and following the following rules:
    - We can only move one top disc at a time
    - No bigger disc can be placed on top of smaller disc
    - Move the discs with smallest number of moves possible

Recursion and Recursive Algorithms

- In this case an elegant solution is obtained as follows:
- Procedure: Move n discs from A to C
  - If n == 1 then move the disc from A to C
  - Else
    - Move n-1 discs from A to B
    - Move 1 disc from A to C
    - Move n-1 discs from B to C
- The move function will then call itself three times

Recursion and Recursive Algorithms

- C++ uses what is known as stack memory in order to manage recursive function calls
- Stack follows LIFO (last in first out)
- Each function call is placed on top of a stack
- The last function to be placed on top of the stack will be the first to be processed
- However stack size is limited and therefore the number of recursive function calls is limited too

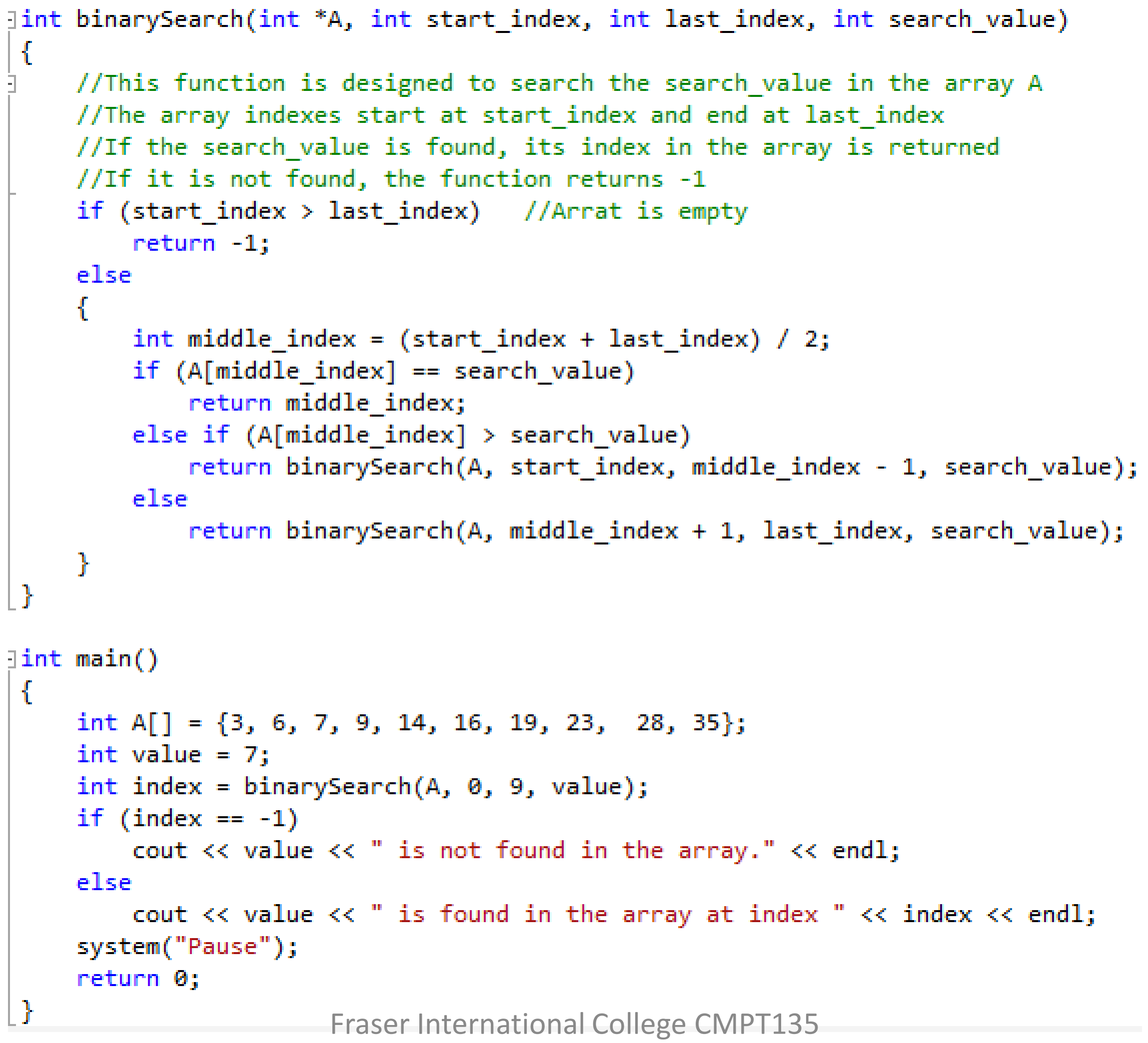
Recursion and Recursive Algorithms

- More Examples
  - **Power Function:-** Given integers x and y such that $x \neq 0$ and $y \geq 0$, $\mathbf{x^y}$ can be written as $\mathbf{x * x^{y-1}}$ with the condition that $\mathbf{x^0}$ is 1.
  - Therefore a power function can be implemented recursively as follows
  - Procedure: pow(x, y)
    - If y == 0, then return 1
    - Else return x * pow(x, y-1)

Divide and Conquer Algorithms
Binary Search

- **Binary Search:-** Given a sorted array we would like to search for a specified value:
  - Procedure: **Binary Search (array, start_index, last_index, value)**
    - If the array is empty return NOT FOUND
    - Else
      - Compute middle index of the array
      - If the element at the middle index is equal to the search value then return FOUND
      - Else if the element at the middle index is greater than the search value then search the first half of the array
      - Else search the second half of the array.

Divide and Conquer Algorithms
Binary Search



```
int binarySearch(int *A, int start_index, int last_index, int search_value)
{
    //This function is designed to search the search_value in the array A
    //The array indexes start at start_index and end at last_index
    //If the search_value is found, its index in the array is returned
    //If it is not found, the function returns -1
    if (start_index > last_index)   //Arrat is empty
        return -1;
    else
    {
        int middle_index = (start_index + last_index) / 2;
        if (A[middle_index] == search_value)
            return middle_index;
        else if (A[middle_index] > search_value)
            return binarySearch(A, start_index, middle_index - 1, search_value);
        else
            return binarySearch(A, middle_index + 1, last_index, search_value);
    }
}

int main()
{
    int A[] = {3, 6, 7, 9, 14, 16, 19, 23,  28, 35};
    int value = 7;
    int index = binarySearch(A, 0, 9, value);
    if (index == -1)
        cout << value << " is not found in the array." << endl;
    else
        cout << value << " is found in the array at index " << index << endl;
    system("Pause");
    return 0;
}
```

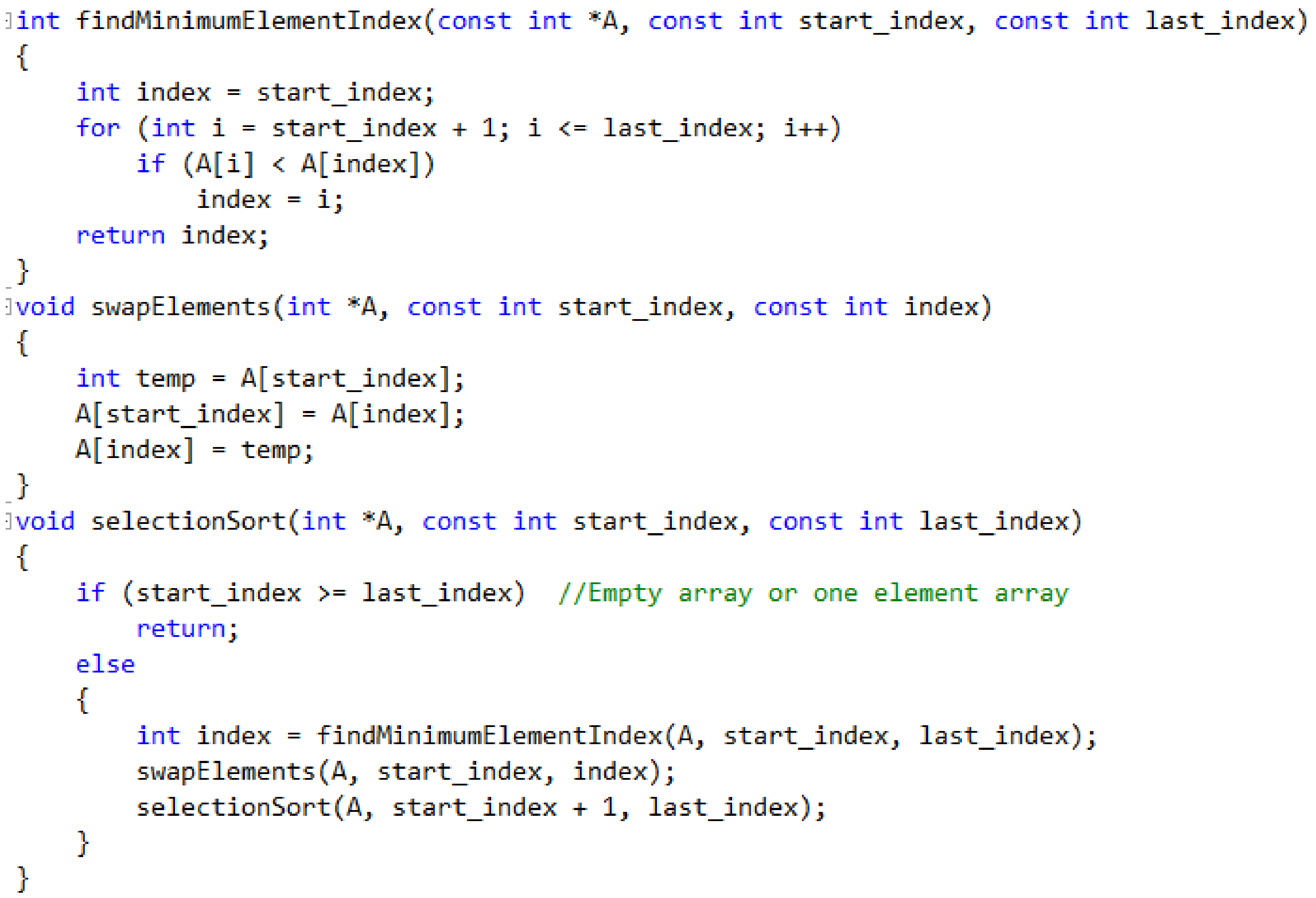
Divide and Conquer Algorithms

Selection, Bubble and Insertion Sorting Algorithms

- Consider selection sort. It sorts a given array **A** with **n** elements at indexes 0,1,2,…n-1 as follows:
  - For each index i = 0, 1, 2,…,n-1 select the minimum element among the A[i], A[i+1],…,A[n-1]
  - Swap that minimum element with element A[i]
- Then a recursive version of the algorithm will be as follows: given an array **A** with a start index **start** and last index **last**
  - If the given array is empty (**start > last**), then there is nothing to sort and we are done
  - Otherwise find the minimum elements among the elements A[start], A[start+1],….,A[last]
  - Swap that minimum element with A[start]
  - Then sort the sub array A[start+1], A[start+2],…,A[last] with the same sorting algorithm

Divide and Conquer Algorithms

Selection, Bubble and Insertion Sorting Algorithms



```
int findMinimumElementIndex(const int *A, const int start_index, const int last_index)
{
    int index = start_index;
    for (int i = start_index + 1; i <= last_index; i++)
        if (A[i] < A[index])
            index = i;
    return index;
}
void swapElements(int *A, const int start_index, const int index)
{
    int temp = A[start_index];
    A[start_index] = A[index];
    A[index] = temp;
}
void selectionSort(int *A, const int start_index, const int last_index)
{
    if (start_index >= last_index)  //Empty array or one element array
        return;
    else
    {
        int index = findMinimumElementIndex(A, start_index, last_index);
        swapElements(A, start_index, index);
        selectionSort(A, start_index + 1, last_index);
    }
}
```

Typical Algorithms and Their Complexities

- Efficiency of algorithms are categorized according to their complexities as follows:

Complexity	**Notation**	**Example**
Constant	$O(1)$	Check one if statement
Logarithmic	$O(\log n)$	Binary Search
Linear	$O(n)$	Sequentional Search
Quasi-linear	**$O(n \log n)$**	**Merge Sort (see below)**
Quadratic	$O(n^2)$	Selection sort
Exponential	$O(2^n)$	Tower's of Hanoi
Factorial	$O(n!)$	Permutation of symbols
n to the n	$O(n^n)$	Solving Sudoku Puzzle

Divide and Conquer Algorithms Merge Sort Algorithm

- Given an array **A** of some comparable elements (say integers), we would like to sort the elements of the array
- An elegant solution would be as follows:
  - Procedure MergeSort(Array)
    - If the array is empty or if it has only one element then there is nothing to do and we are done
    - Else
      - ➢ First sort the first half
      - ➢ Then sort the second half
      - ➢ Then Merge the two halves

Divide and Conquer Algorithms Merge Sort Algorithm

- Interestingly, in order to sort the first half; we use the same sorting procedure: MergeSort
- Moreover, in order to sort the second half; we use the same sorting procedure: MergeSort
- Example suppose the array to be sorted is

  [3, 2, 8, 5, 1, 6]
- Then we sort the first half to get [2, 3, 8]
- Then we sort the second half to get [1, 5, 6]

Divide and Conquer Algorithms Merge Sort Algorithm

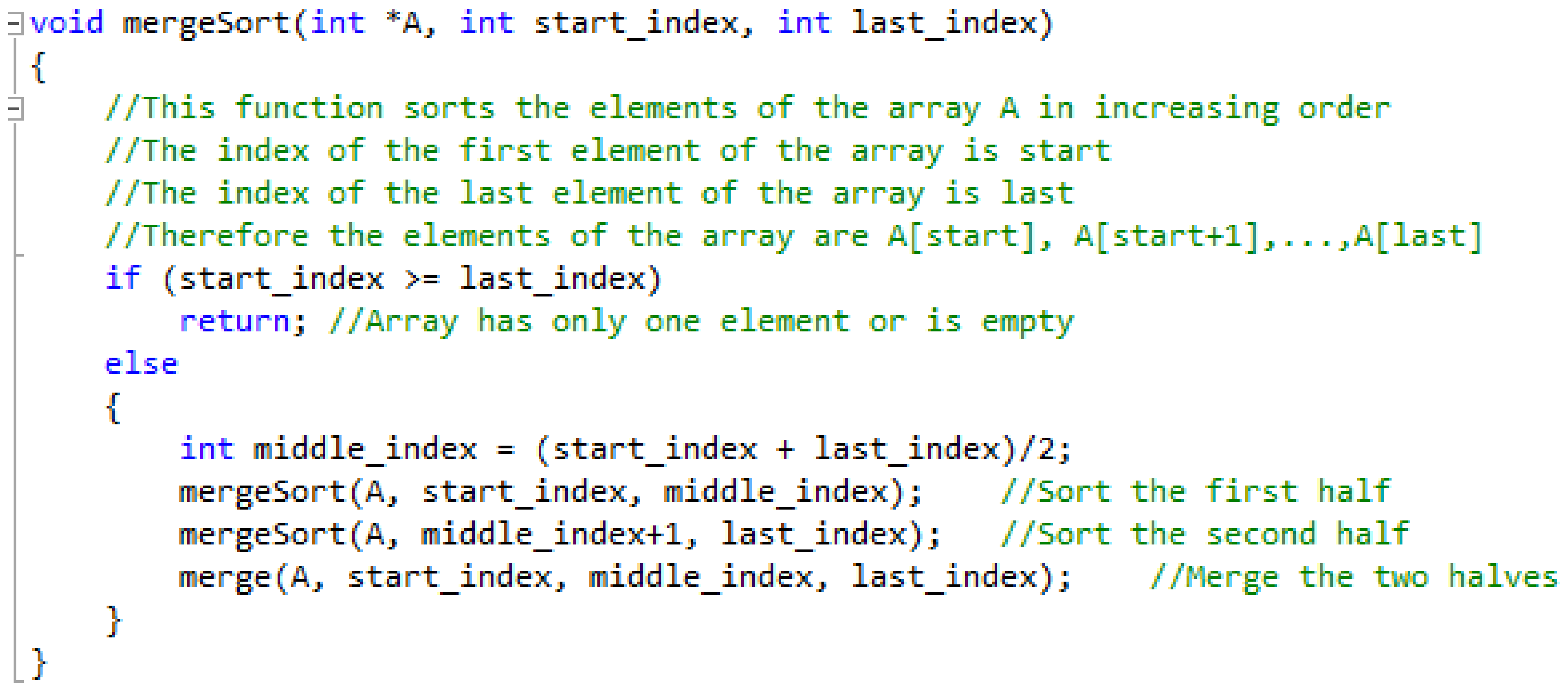
- Now we merge these two sub arrays
- In order to do so, we create a temporary array (call it **temp**) where we are going to put our result and three index variables i, j and k such that indexes i and j start at the beginning of the two sub arrays while index k starts at 0
- Then move the smaller of the elements A[i] or A[j] to **temp[k]** and increment the indexes i or j and k by 1
- If any of the indexes i or j exceeds the last element in its sub array then copy all the remaining elements from the other sub array to the merged array

Divide and Conquer Algorithms
Merge Sort Algorithm

- In order to trace the start index and last index of the sub arrays in our merge function, we should also provide the function the starting index and last index of the array to be sorted
- Moreover an array is empty if its start index is larger than its last index
- Therefore we may write our mergeSort function to sort an array of integers as follows:

Divide and Conquer Algorithms Merge Sort Algorithm

- **Merge Sort Algorithm**

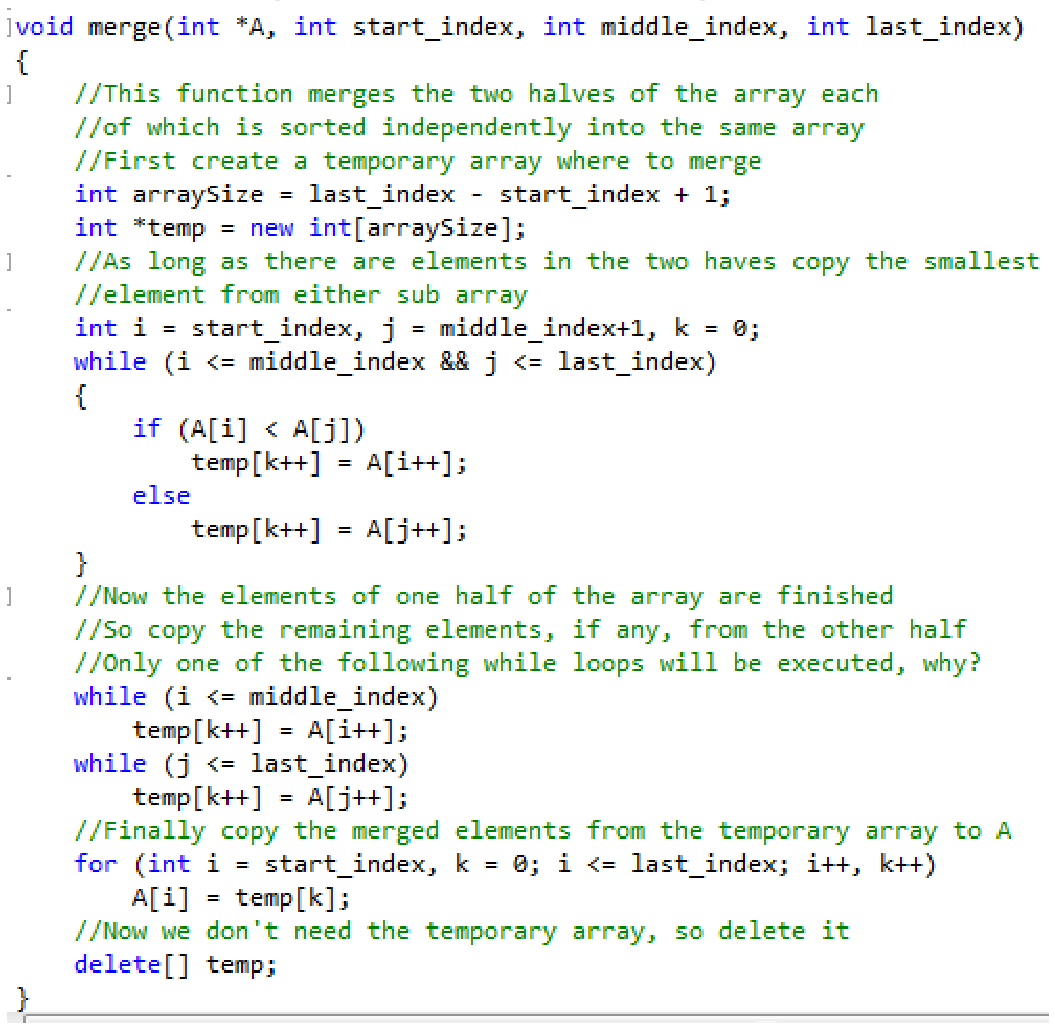


```cpp
void mergeSort(int *A, int start_index, int last_index)
{
    //This function sorts the elements of the array A in increasing order
    //The index of the first element of the array is start
    //The index of the last element of the array is last
    //Therefore the elements of the array are A[start], A[start+1],...,A[last]
    if (start_index >= last_index)
        return; //Array has only one element or is empty
    else
    {
        int middle_index = (start_index + last_index)/2;
        mergeSort(A, start_index, middle_index);    //Sort the first half
        mergeSort(A, middle_index+1, last_index);   //Sort the second half
        merge(A, start_index, middle_index, last_index);    //Merge the two halves
    }
}
```

Divide and Conquer Algorithms Merge Sort Algorithm

- We decide to write the merging process in a separate function named **merge** for code clarity purposes
- The merge function will use a temporary array of the same size as the array being merged in order to merge into the temporary array
- Of course we could merge within the same array too although a little bit involved

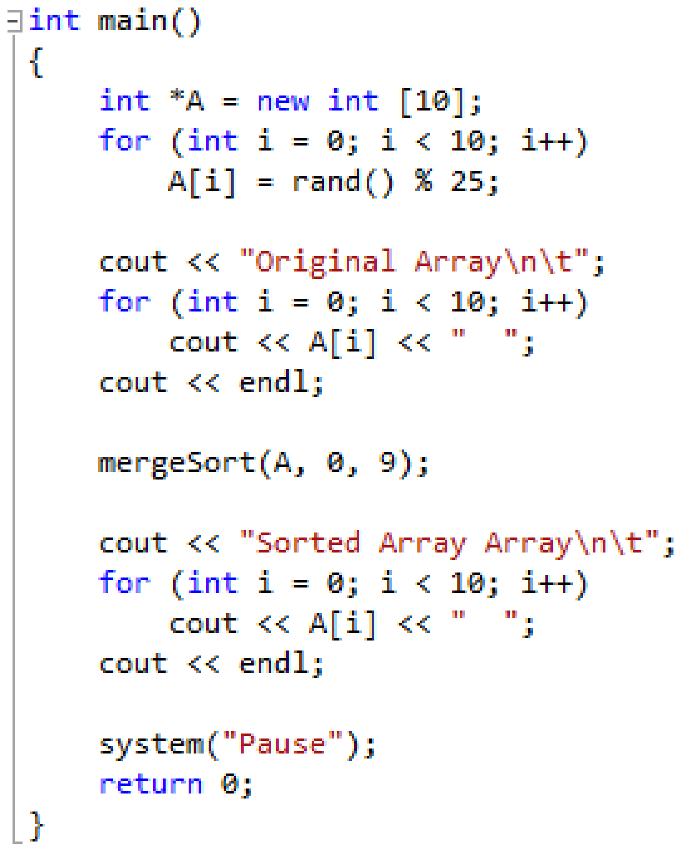
Divide and Conquer Algorithms
Merge Sort Algorithm



```
void merge(int *A, int start_index, int middle_index, int last_index)
{
    //This function merges the two halves of the array each
    //of which is sorted independently into the same array
    //First create a temporary array where to merge
    int arraySize = last_index - start_index + 1;
    int *temp = new int[arraySize];
    //As long as there are elements in the two haves copy the smallest
    //element from either sub array
    int i = start_index, j = middle_index+1, k = 0;
    while (i <= middle_index && j <= last_index)
    {
        if (A[i] < A[j])
            temp[k++] = A[i++];
        else
            temp[k++] = A[j++];
    }
    //Now the elements of one half of the array are finished
    //So copy the remaining elements, if any, from the other half
    //Only one of the following while loops will be executed, why?
    while (i <= middle_index)
        temp[k++] = A[i++];
    while (j <= last_index)
        temp[k++] = A[j++];
    //Finally copy the merged elements from the temporary array to A
    for (int i = start_index, k = 0; i <= last_index; i++, k++)
        A[i] = temp[k];
    //Now we don't need the temporary array, so delete it
    delete[] temp;
}
```

Divide and Conquer Algorithms Merge Sort Algorithm

- A test main program might look like:



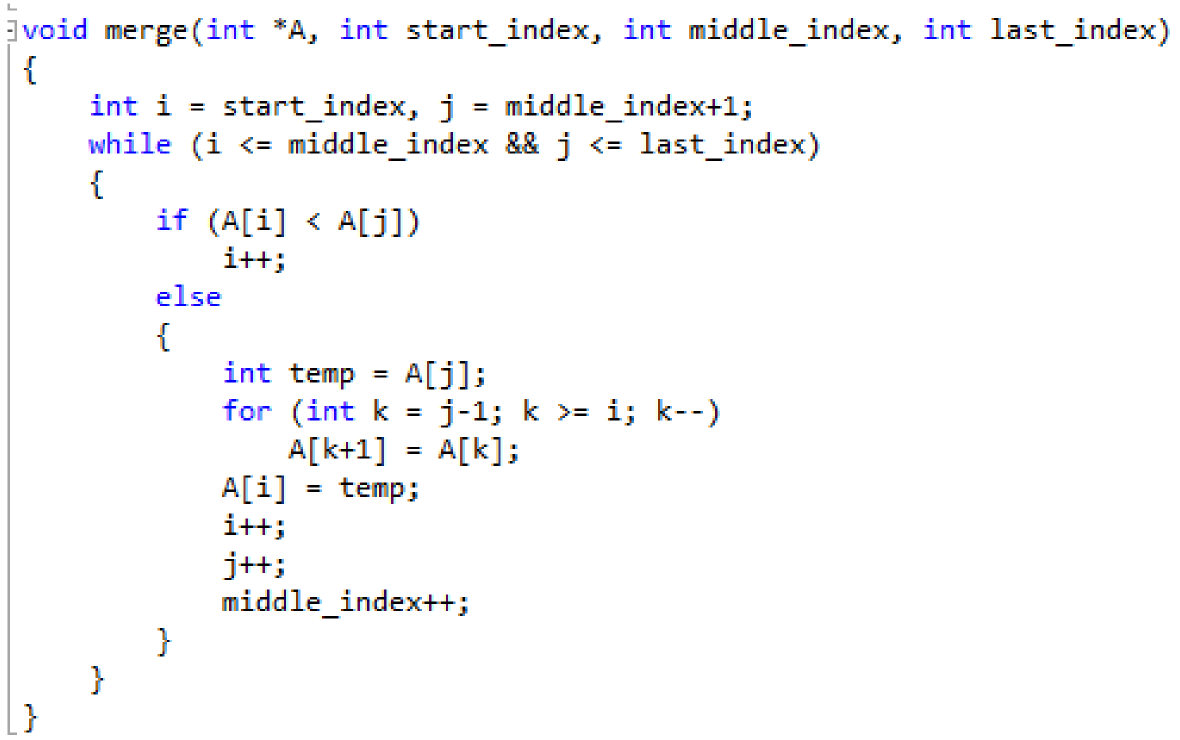
```
int main()
{
    int *A = new int [10];
    for (int i = 0; i < 10; i++)
        A[i] = rand() % 25;

    cout << "Original Array\n\t";
    for (int i = 0; i < 10; i++)
        cout << A[i] << "  ";
    cout << endl;

    mergeSort(A, 0, 9);

    cout << "Sorted Array Array\n\t";
    for (int i = 0; i < 10; i++)
        cout << A[i] << "  ";
    cout << endl;

    system("Pause");
    return 0;
}
```

Divide and Conquer Algorithms Merge Sort Algorithm

- The merge sort algorithm is a very elegant and efficient algorithm
- The complexity of the algorithm is **O(n log n)**
- This is much better than the quadratic sorting algorithms: bubble sort, selection sort and insertion sort
- Challenge:- Re-write the merge function such that it does not create a temporary array for merging purposes. Instead do the merging process within the same array that is being merged.

Merging within the same array

- Look at the following code snapshot that is purported to merge two sub arrays within the same array and convince yourself indeed it does what is expected from it.



```
void merge(int *A, int start_index, int middle_index, int last_index)
{
    int i = start_index, j = middle_index+1;
    while (i <= middle_index && j <= last_index)
    {
        if (A[i] < A[j])
            i++;
        else
        {
            int temp = A[j];
            for (int k = j-1; k >= i; k--)
                A[k+1] = A[k];
            A[i] = temp;
            i++;
            j++;
            middle_index++;
        }
    }
}
```