CSCI 2170 **Lecture Notes (16)**

Sorting Aglorithms

```
void BubbleSort(dataType A[], int N)
{
   bool Sorted = false;  // false when swaps occur

   for (int Pass = 1; (Pass < N) && !Sorted; ++Pass)
   {  // Invariant: A[N+1-Pass..N-1] is sorted
      //           and > A[0..N-Pass]
      Sorted = true;  // assume sorted
      for (int Index = 0; Index < N-Pass; ++Index)
      {  // Invariant: A[0..Index-1] <= A[Index]
         int NextIndex = Index + 1;
         if (A[Index] > A[NextIndex])
         {  // exchange items
            Swap(A[Index], A[NextIndex]);
            Sorted = false;  // signal exchange
         }  // end if
      }  // end for

      // Assertion: A[0..N-Pass-1] < A[N-Pass]
   }  // end for
}  // end BubbleSort

void Swap(dataType& X, dataType& Y)
{
   dataType Temp = X;
   X = Y;
   Y = Temp;
}  // end Swap
```

```
void SelectionSort(dataType A[], int N)
{
  // Last = index of the last item in the subarray of
  //       items yet to be sorted,
  // L = index of the largest item found

  for (int Last = N-1; Last >= 1; --Last)
  {  // Invariant: A[Last+1..N-1] is sorted and >
     // A[0..Last]

     // select largest item in A[0..Last]
     int L = IndexOfLargest(A, Last+1);

     // swap largest item A[L] with A[Last]
     Swap(A[L], A[Last]);
  }  // end for
}  // end SelectionSort

int IndexOfLargest(const dataType A[], int Size)
{
  int IndexSoFar = 0;  // index of largest item
                       // found so far

  for (int CurrentIndex = 1; CurrentIndex < Size;
                            ++CurrentIndex)
  {  // Invariant: A[IndexSoFar] >=
     //           A[0..CurrentIndex-1]
     if (A[CurrentIndex] > A[IndexSoFar])
        IndexSoFar = CurrentIndex;
  }  // end for

  return IndexSoFar;  // index of largest item
}  // end IndexOfLargest
```

```
void InsertionSort(dataType A[], int N)
{
   for (int Unsorted = 1; Unsorted < N; ++Unsorted)
   {  // Invariant: A[0..Unsorted-1] is sorted

      // find the right position (Loc) in
      // A[0..Unsorted] for A[Unsorted], which is the
      // first item in the unsorted region;
      // shift, if necessary, to make room
      dataType NextItem = A[Unsorted];
      int Loc = Unsorted;

      for (;(Loc > 0) && (A[Loc-1] > NextItem); --Loc)
         // shift A[Loc-1] to the right
         A[Loc] = A[Loc-1];
      // Assertion: A[Loc] is where NextItem belongs

      // insert NextItem into Sorted region
      A[Loc] = NextItem;
   }  // end for
}  // end InsertionSort
```

```
void Mergesort(dataType A[], int F, int L)
// ---------------------------------------------------------
// Sorts the items in an array into ascending order.
// Precondition: A[F..L] is an array.
// Postcondition: A[F..L] is sorted in ascending order.
// Calls: Merge.
// ---------------------------------------------------------
{
   if (F < L)
   {  // sort each half
      int Mid = (F + L)/2;    // index of midpoint
      Mergesort(A, F, Mid);   // sort left half A[F..Mid]
      Mergesort(A, Mid+1, L); // sort right half A[Mid+1..L]

      // merge the two halves
      Merge(A, F, Mid, L);
   }  // end if
}  // end Mergesort

void Merge(dataType A[], int F, int Mid, int L)
{
   dataType TempArray[MAX_SIZE];    // temporary array

   // initialize the local indexes to indicate the subarrays
   int First1 = F;        // beginning of first subarray
   int Last1  = Mid;      // end of first subarray
   int First2 = Mid + 1;  // beginning of second subarray
   int Last2  = L;        // end of second subarray

   // while both subarrays are not empty, copy the
   // smaller item into the temporary array
   int Index = First1;    // next available location in
                          // TempArray
   for (; (First1 <= Last1) && (First2 <= Last2); ++Index)
   {  // Invariant: TempArray[First1..Index-1] is in order
      if (A[First1] < A[First2])
      {  TempArray[Index] = A[First1];
         ++First1;
      }
      else
      {  TempArray[Index] = A[First2];
         ++First2;
      }  // end if
   }  // end for

   // finish off the nonempty subarray

   // finish off the first subarray, if necessary
   for (; First1 <= Last1; ++First1, ++Index)
      // Invariant: TempArray[First1..Index-1] is in order
      TempArray[Index] = A[First1];
```

```
   // finish off the second subarray, if necessary
   for (; First2 <= Last2; ++First2, ++Index)
      // Invariant: TempArray[First1..Index-1] is in order
      TempArray[Index] = A[First2];

   // copy the result back into the original array
   for (Index = F; Index <= L; ++Index)
      A[Index] = TempArray[Index];
}  // end Merge
```

```
void Quicksort(dataType A[], int F, int L)
// ---------------------------------------------------------
// Sorts the items in an array into ascending order.
// Precondition: A[F..L] is an array.
// Postcondition: A[F..L] is sorted.
// Calls: Partition.
// ---------------------------------------------------------
{
   int PivotIndex;

   if (F < L)
   {  // create the partition: S1, Pivot, S2
      Partition(A, F, L, PivotIndex);

      // sort regions S1 and S2
      Quicksort(A, F, PivotIndex-1);
      Quicksort(A, PivotIndex+1, L);
   }  // end if
}  // end Quicksort

void ChoosePivot(dataType A[], int F, int L);
// ---------------------------------------------------------
// Chooses a pivot for quicksortÕs partition algorithm and swaps it with the first item in an array.
// Precondition: A[F..L] is an array; F <= L.
// Postcondition: A[F] is the pivot.
// ---------------------------------------------------------
void Partition(dataType A[], int F, int L, int& PivotIndex)
{
   ChoosePivot(A, F, L);      // place pivot in A[F]
   dataType Pivot = A[F];     // copy pivot

   // initially, everything but pivot is in unknown
   int LastS1 = F;            // index of last item in S1
   int FirstUnknown = F + 1;  // index of first item in unknown

   // move one item at a time until unknown region is empty
   for (; FirstUnknown <= L; ++FirstUnknown)
   {  // move item from unknown to proper region
      if (A[FirstUnknown] < Pivot)
      {  // item from unknown belongs in S1
         ++LastS1;
         Swap(A[FirstUnknown], A[LastS1]);
      }  // end if

      // else item from unknown belongs in S2
   }  // end for

   // place pivot in proper position and mark its location
   Swap(A[F], A[LastS1]);
   PivotIndex = LastS1;
}  // end Partition
```

```
RadixSort(A, N, d)
// Sort N d-digit integers in the array A

  for (J=d down to 1)
  {
     Initialize 10 groups to empty
     Initialize a counter for each group to 0
     for (I = 0 through N-1)
     {
          K=Jth digit of A[I]
          Place A[I] at the end of group K
          Increase Kth counter by 1
      }

      Replace the items in A with all the items in group 0,
      Followed by all the items in group 1 and so on.
   }  // end for J
```