MergeSort

**void** merge(**int** A[], **int** a, **int** mid, **int** b) {

// PRE: the array portions A[a..mid-1]

// and A[mid..b] are sorted

// POST: The array portion A[a..b] are sorted

**if** (a <= b && mid >= a && mid <= b) {

**int**[] tA= **new** **int**[b-a+1];

**int** i= a; **int** j= mid; **int** ti= 0;

// INVARIANT for all the loops:

// tA[0.. (i-a) + (j-mid) -1] is the sorted merge

// of A[a..i-1] and A[mid..j-1]

**while**(i <mid && j < b+1) {

**if** (A[i] > A[j]) {tA[ti]= A[j]; j++;}

**else** {tA[ti]= A[i]; i++;}

ti++;

}

**if** (i < mid) {

**for** (; i< mid; i++){

tA[ti]= A[i]; ti++;

}

}

**else** {

**for** (; j< b+1; j++){

tA[ti]= A[j]; ti++;

}

}

// copy back to A

**for**(**int** k= a; k<= b; k++){

A[k]= tA[k-a];

}

}

}

**void** mergeSort(**int** A[], **int** first, **int** last){

// POST: sorts the array portion A[first..last]

**if** (first < last) { // just swap the nodes;

mergeSort (A, first, (first+last)/2);

mergeSort (A, (first+last)/2 + 1, last);

// PRE for MERGE: the array portions A[first..(first+last)/2]

// and A[(first+last)/2+1..last] are sorted

merge(A, first, (first+last)/2+1, last);

}

}

Quick Sort

**int** partition(**int** A[], **int** first, **int** last) {

// Take A[first] as the item to be placed in correct

// sorted order. Set pivotIndex to that item and return it.

**int** p= A[first]; // Initialisation to satisfy the invariant.

**int** LastS1= first;

**int** FirstUnknown= first + 1;

**int** temp;

**while** (FirstUnknown <= last) {

// INVARIANT: The items in myArray[first+1..LastS1]

// are all strictly less than p,

// and the items in myArray[LastS1+1.. FirstUnknown-1] are at least p.

**if** (A[FirstUnknown] < p) { // Case: need to add to S1

LastS1++; // Increment LastS1

temp= A[FirstUnknown];

A[FirstUnknown]= A[LastS1];

A[LastS1]= temp;

FirstUnknown++;

}

**else** {

FirstUnknown++;

} // Case: Need to add to S2

}

temp= A[first];

A[first]= A[LastS1];

A[LastS1]= temp;

FirstUnknown++;

**return** LastS1;

}

**void** swap(**char** A[], **int** i, **int** j) {

**char** temp= A[i];

A[i]= A[j];

A[j]= temp;

}

**void** quicksort (**int** A[], **int** first, **int** last) {

**if** (first < last) {

// Always choose p to be A[first].

**int** pivotIndex=first;

pivotIndex= partition( A, first, last); // update pivotIndex

quicksort(A, first, pivotIndex-1);

quicksort(A, pivotIndex +1, last);

}

}

Bucket Sort

**void** bucketSort(**int** A[], **int** Num[]) {

// For simplicity we assume that A consists of integers, 0... Num.length-1

// PRE: The items in A must have value no more that Num.length-1

**int** n= A.length;

**for** (**int** i= 0; i < n; i++){

// INVARIANT: # entries value k in A[0..i-1] is Num[k].

// Declare a structure for the ÒbucketsÓ

**int** k=A[i]; Num[k]++;} // Increment k'th bucket

**int** m= 0;

**int** sofar= 0;

**for** (**int** j = 0; j < Num.length; j++){

// Invariant: A[0..sofar] is sorted,

// containing all the elements up to j

**for**(; m< sofar + Num[j]; m++)

A[m]= j;

sofar= m;

}

}

Radix Sort

void radixSort(int A[]; int n; int d){

// PRE: A is an array of d-digit binary integers; n is the length of A;

// POST: A is sorted

for(int j=0; j<d; j++){

int G0[n]; int count0=0;

int G1[n]; int count1=0;

for (int i=0; i<n; i++){

int k=digit(A[i],j);

switch (k){

case 0: {G0[count0]=A[i]; count0++; break)}

case 1: {G1[count1]=A[i]; count1++; break)}

}

}

int y = 0

for (int m=0; m<count0; m++){

A[y]=G0[m]; y++

}

for (int m=0; m<count1; m++){

A[y]=G1[m]; y++;

}

}

}

int digit(int a; int b){

// PRE: a is a binary integer, and b is at least 0

// POST: returns the b'th digit of a