Ch 9 Sorting (DS)

Internal Sort: Insertion Sort, Bubble Sort, Quick Sort, and Heap Sort. (RAM)

External Sort: Merge Sort.(disks, taps, ROMs)

Data compositions: files of records containing keys

File: collection of records

Record: collection of fields

Field: collection of characters

Key: uniquely identifies a record in a file, used to control the sort

Selection Sort

Select successive elements in ascending order, place them into their proper order.

Find the smallest, exchange it with the first, find the second smallest, exchange it with the second, etc.

```
void selection_sort(int array[], int n){
  int i, j, min, tmp;
  for (i = 0; i < n; i++){
    min = i;
    for (j = 0; j < n; j++){
      if (array[j] < array[min]) min = j;
    }
    tmp = array[min];
    array[min] = array[i];
    array[i] = tmp;
  }
}
```

Insertion Sort

Take elements one by one, insert them by proper position among these already taken and sorted in a new collection, repeat until done.

```
void insertion_sort(int array[], int n){
  int i, j, tmp;
  for (i = 0; i < n; i++){
    tmp = array[i];
    for (j = 0; j < n; j++){
      array[j] = array[j-1];
    }
    array[j] = tmp;
  }
}
```

Bubble Sort

Compare two elements in the neighbor, exchange them if not in proper order. The greatest element moves to the most right, repeat except the most right element until done.

```
void bubble_sort(int array[], int n){
  int i, j, tmp;
  for (i = n; i < 0; i--){
    for (j = 1; j < i; j++){
      if (array[j-1] > array[j]){
        tmp = array[j-1];
        array[j-1] = array[j];
        array[j] = tmp;
      }
    }
  }
}
```

Quick Sort

Divide and Conquer

Choose a pivotal (central) element, move elements smaller than that to the left and greater to the right. Repeat with smaller lists until done.

```
void quick_sort(int array[]. int left, int right){
  int i;
  if (right > left){
    i = partition(left, right);
    quick_sort(array, left, i-1);
    quick_sort(array, i+1, right);
  }
}
// partition(left, right) =>
// when the element array[i] is placed in its final place, all elements of array[left], ..., array[i-1] are smaller than or equal to array[i], and all elements of array[i+1], ..., array[right] are larger than or equal to array[i]. That is, array[j] <= array[i] for j < i and array[j] >= array[i] for j > i


void quick_sort(int array[]. int left, int right){
  int v, i, j, tmp;
  if (right > left){
    v = array[right];
    i = left - 1;
    j = right;
    for (;;){
      while (array[++i] < v);
      while (array[--j] > v);
      if (i >= j) break;
      tmp = array[i];
      array[i] = array[j];
      array[j] = tmp;
    }
    tmp = array[i];
    array[i] = array[right];
    array[right] = tmp;
    quick_sort(array, left, i-1);
    quick_sort(array, i+1, right);
  }
}
```

Heap Sort

```
void down_heap(int array[], int n, int x){
  int i, v;
  v = array[x];
  while (x <= n/2){
    i = x+x;
    if (i < n && array[i] < array[i+1]) i++;
    if (v >= array[i]) break;
    array[x] = array[i];
    x = i;
  }
  array[x] = v;
}


void heap_sort(int array[]. int n){
  int x, tmp;
  for (x = n/2; x >= 0; x--){
    down_heap(array, n, x);
  }
  while (n > 0){
    tmp = array[0];
    array[0] = array[n];
    array[n] = tmp;
    down_heap(array, --n, 0);
  }
}
```

Merge Sort

```
void merge_sort(int a[], int l, int r){
  int i, j, k, m;
  if (r > l){
    m = (r+l)/2;
    merge_sort(a, l, m);
    merge_sort(a, m+l, r);
```

```
    for (i = m+1; i > l; i--) b[i-1] = a[i-1];
    for (j = m; j < r; j++) b[r+m-j] = a[j+1];
    for (k = l; k <= r; k++)
      a[k] = (b[i] < b[j]) ? b[i++] : b[j--];
  }
}
```

Shell Sort

improvement of Insertion Sort

```
void shell_sort(int array[], int n){
  int i, j, k, tmp;
  for (k = 1; k < n/9; k = 3*k+1);
  for (; k > 0; k /= 3){
    for (i = k; i < n; i++){
      j = i;
      while (j >= k && array[j-k] > array[j]){
        tmp = array[j];
        array[j] = array[j-k];
        array[j-k] = tmp;
        j -= k;
      }
    }
  }
}
```

Radix Sort

Scans from left to find a key which starts with bit 1, scans from the right to find a key which starts with bit 0, then exchange, continue until done.

```
void radix_exchange(int array[], int l; int r, int b){
  int i, j, tmp;
  if (r > l && b >= 0){
    i = l;
    j = r;
    while (j != i){
      while (bits(array[i], b, 1) == 0 && i < j) i++;
      while (bits(array[j], b, 1) != 0 && j < i) j--;
      tmp = array[i];
      array[i] = array[j];
      array[j] = tmp;
    }
    if (bits(array[r], b, 1) == 0) j++;
    radix_exchange(array, l, j-1, b-1);
    radix_exchange(array, j, r, b-1);
  }
}
```

Algorithms	Performance	Comments
Selection Sort	N^2	good for small and partially sorted data
Insertion Sort	N^2	good for almost sorted data
Bubble Sort	N^2	good for $N < 100$
Quick Sort	$N \log N$	excellent
Heap Sort	$N \log N$	excellent
Merge Sort	$\log N$	good for external sort
Shell Sort	$N^{1.5}$	good for medium N
Radix Sort	$N \log N$	excellent