**CS20L: Information Structures**

**Semester II, 2004**

### Sorting

One of the fundamental problems of computer science is ordering a list of items. There's a plethora of solutions to this problem, known as sorting algorithms. Some sorting algorithms are simple and intuitive, such as the bubble sort. Others, such as the quick sort are extremely complicated, but produce lightning-fast results.

**Definitions:**

Sorting algorithms are divided into two categories: internal and external sorts.

**Internal Sort:**

Any sort algorithm which uses main memory exclusively during the sort. This assumes high-speed random access to all memory.

**External Sort:**

Any sort algorithm which uses external memory, such as tape or disk, during the sort.

*Note: Algorithms may read the initial values from magnetic tape or write sorted values*

*to disk, but this is not using external memory during the sort. Note that even though virtual memory may mask the use of disk, sorting sets of data much larger than main memory may be much faster using an explicit external sort.*

Sort Stable

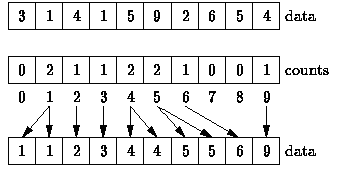
A sort algorithm is said to be “stable” if multiple items which compare as equal will stay in the same order they were in after a sort.

**Examples of Internal Sort Techniques**

1. **Bucket Sort**

Bucket sort is possibly the simplest distribution sorting algorithm. The essential requirement is that the size of the universe from which the elements to be sorted are drawn is a small, fixed constant, say *m*.

For example, suppose that we are sorting elements drawn from **{0, 1, . . ., m-1}**, i.e., the set of integers in the interval **[0, *m*-1]**. Bucket sort uses ***m*** counters. The ***ith*** counter keeps track of the number of occurrences of the ***ith*** element of the universe. The figure below illustrates how this is done.



**Figure:** Bucket Sorting

In the figure above, the universal set is assumed to be **{0, 1, . . ., 9}**. Therefore, ten counters are required-one to keep track of the number of zeroes, one to keep track of the number of ones, and so on. A single pass through the data suffices to count all of the elements. Once the counts have been determined, the sorted sequence is easily obtained. E.g., the sorted sequence contains no zeroes, two ones, one two, and so on.

##### Program Implementation

**void bucketSort(dataElem array[], int array\_size)**

**{**

**int i, j;**

**dataElem count[array\_size];**

**for(i =0; i < array\_size; i++)**

**count[i] = 0;**

**for(j =0; j < array\_size; j++)**

**++count[array[j]];**

**for(i =0, j=0; i < array\_size; i++)**

**for(; count[i]>0; --count[i])**

**array[j++] = i;**

**}**

# 2. Selection Sorting

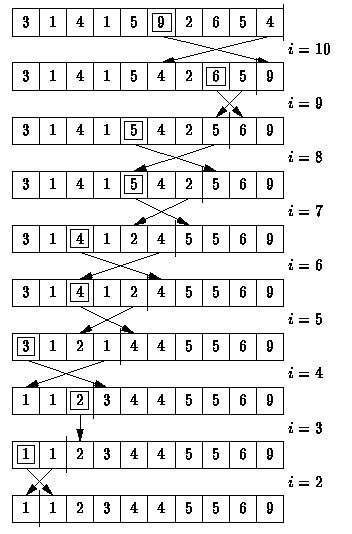
The selection sort algorithms constructs the sorted sequence one element at a time by adding elements to the sorted sequence in order. At each step, the next element to be added to the sorted sequence is selected from the remaining elements.

Because the elements are added to the sorted sequence in order, they are always added at one end. This is what makes selection sorting different from insertion sorting. In insertion sorting elements are added to the sorted sequence in an arbitrary order. Therefore, the position in the sorted sequence at which each subsequent element is inserted is arbitrary.

Both selection and insertion sorts sort the arrays in place. Consequently, the sorts are implemented by exchanging array elements. Nevertheless, selection differs from exchange sorting because at each step we select the next element of the sorted sequence from the remaining elements and then we move it into its final position in the array by exchanging it with whatever happens to be occupying that position.

At each step of the algorithm, a linear search of the unsorted elements is made in order to determine the position of the largest remaining element. That element is then moved into the correct position of the array by swapping it with the element which currently occupies that position.

For example, in the first step shown in the Figure below, a linear search of the entire array reveals that 9 is the largest element. Since 9 is the largest element, it belongs in the last array position. To move it there, we swap it with the 4 that initially occupies that position. The second step of the algorithm identifies 6 as the largest remaining element and moves it next to the 9. Each subsequent step of the algorithm moves one element into its final position.



**Pros:** Simple and easy to implement.  
**Cons:** Inefficient for large lists, so similar to the more efficient insertion sort that the insertion

sort should be used in its place.

**Selection Sort Program Implementation**

**void selectionSort( dataElem array[ ], int array\_size)**

**{**

**int i, j;**

**int min, temp;**

**for (i = 0; i < array\_size-1; i++)**

**{**

**min = i;**

**for (j = i+1; j < array\_size; j++)**

**{**

**if (array [j] < array [min])**

**min = j;**

**}**

**temp = array [i];**

**array [i] = array [min];**

**array [min] = temp;**

**}**

**}**

#### 3. Insertion Sorting

The insertion sort works just like its name suggests - it inserts each item into its proper place in the final list. The simplest implementation of this requires two list structures - the source list and the list into which sorted items are inserted. To save memory, most implementations use an in-place sort that works by moving the current item past the already sorted items and repeatedly swapping it with the preceding item until it is in place.

**Pros:** Relatively simple and easy to implement.

**Cons:** Inefficient for large lists.

**Insertion Sort Program Implementation**

**void insertionSort(int array[], int array\_size)**

**{**

**int i, j, index;**

**for (i=1; i < array\_size; i++)**

**{**

**index = array[i];**

**j = i;**

**while ((j > 0) && (array[j-1] > index))**

**{**

**array[j] = array[j-1];**

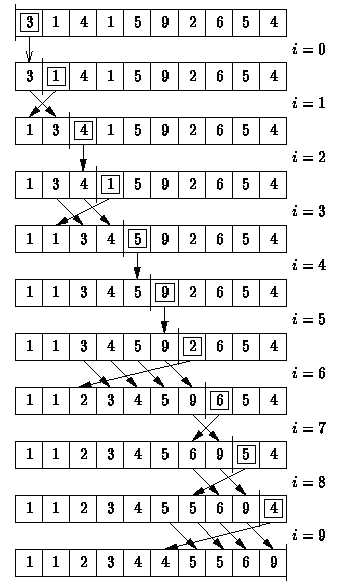
**j = j - 1;**

**}**

**array[j] = index;**

**}**

**}**

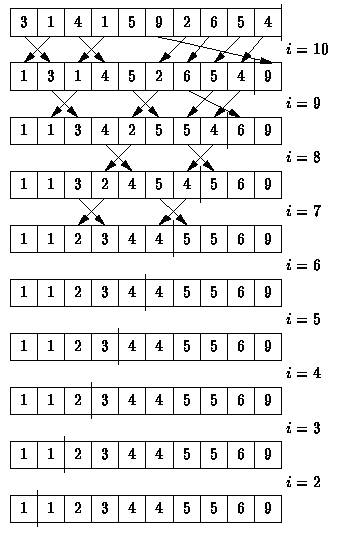
  
 **Figure:** Insertion Sorting

## 4. Bubble Sort

The bubble sort is the oldest and simplest sort in use. Unfortunately, it's also the slowest.

The bubble sort works by comparing each item in the list with the item next to it, and swapping them if required. The algorithm repeats this process until it makes a pass all the way through the list without swapping any items (in other words, all items are in the correct order). This causes larger values to "bubble" to the end of the list while smaller values "sink" towards the beginning of the list.

**Pros:** Simple and easy to implement.  
**Cons:** Horribly inefficient.

   
 **Figure:** Bubble Sorting

**Bubble Sort Program Implementation**

**void bubbleSort(int array[], int array\_size)**

**{**

**int i, j, temp;**

**for (i = (array\_size - 1); i >= 0; i--)**

**{**

**for (j = 1; j <= i; j++)**

**{**

**if (array[j-1] > array[j])**

**{**

**temp = array[j-1];**

**array[j-1] = array[j];**

**array[j] = temp;**

**}**

**}**

**}**

**}**

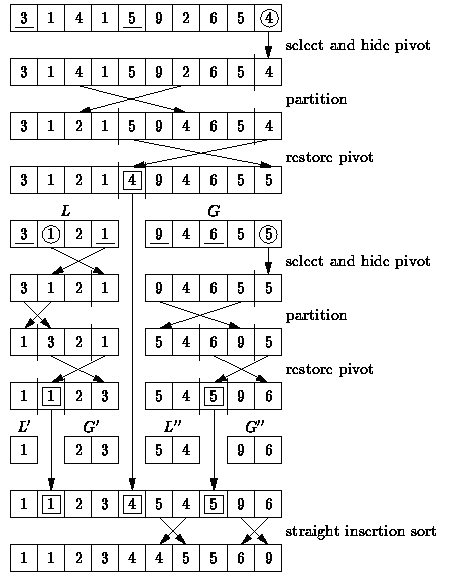
## 5. Quicksort

The quick sort is an in-place, divide-and-conquer, massively recursive sort. The algorithm is simple in theory, but very difficult to put into code (computer scientists tied themselves into knots for years trying to write a practical implementation of the algorithm, and it still has that effect on university students).

The recursive algorithm consists of four steps:

1. If there is one or less element in the array to be sorted, return immediately.
2. Pick an element in the array to serve as a "pivot" point. (Usually the left-most element in the array is used.)
3. Split the array into two parts - one with elements larger than the pivot and the other with elements smaller than the pivot.
4. Recursively repeat the algorithm for both halves of the original array.

**Pros:** Extremely fast.  
**Cons:** Very complex algorithm, massively recursive.

  
 **Figure:** “Quick” Sorting

### Quick Sort Program Implementation

**void q\_sort(int array[], int left, int right)**

**{**

**int pivot, l\_hold, r\_hold;**

**l\_hold = left;**

**r\_hold = right;**

**pivot = array[left];**

**while (left < right)**

**{**

**while ((array[right] >= pivot) && (left < right))**

**right--;**

**if (left != right)**

**{**

**array[left] = array[right];**

**left++;**

**}**

**while ((array[left] <= pivot) && (left < right))**

**left++;**

**if (left != right)**

**{**

**array[right] = array[left];**

**right--;**

**}**

**}**

**array[left] = pivot;**

**pivot = left;**

**left = l\_hold;**

**right = r\_hold;**

**if (left < pivot)**

**q\_sort(array, left, pivot-1);**

**if (right > pivot)**

**q\_sort(array, pivot+1, right);**

**}**

**void quickSort(int array[], int array\_size)**

**{**

**q\_sort(array, 0, array\_size - 1);**

**}**

#### 6. Heap Sort

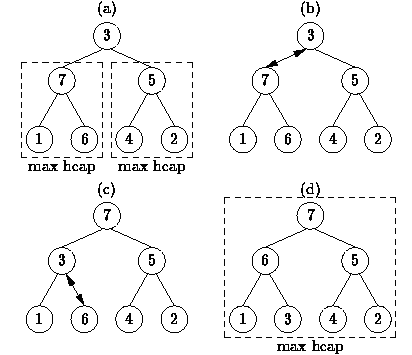
The heap sort is the slowest of the fast sorting algorithms, but unlike algorithms such as the merge and quick sorts it does not require massive recursion or multiple arrays to work. This makes it the most attractive option for *very* large data sets of millions of items.

The heap sort works as it name suggests - it begins by building a heap out of the data set, and then removing the largest item and placing it at the end of the sorted array. After removing the largest item, it reconstructs the heap and removes the largest remaining item and places it in the next open position from the end of the sorted array. This is repeated until there are no items left in the heap and the sorted array is full. Elementary implementations require two arrays - one to hold the heap and the other to hold the sorted elements.

To do an in-place sort and save the space the second array would require, the algorithm below “cheats” by using the same array to store both the heap and the sorted array. Whenever an item is removed from the heap, it frees up a space at the end of the array that the removed item can be placed in.

**Pros:** In-place and non-recursive, making it a good choice for extremely large data sets.  
**Cons:** Slower than the merge and quick sorts.

.

  
 **Figure:** Combining Heaps by Percolating Values

**void siftDown(int array[], int root, int bottom)**

**{**

**int done, maxChild, temp;**

**done = 0;**

**while ((root\*2 <= bottom) && (!done))**

**{**

**if (root\*2 == bottom)**

**maxChild = root \* 2;**

**else if (array[root \* 2] > array[root \* 2 + 1])**

**maxChild = root \* 2;**

**else**

**maxChild = root \* 2 + 1;**

**if (array[root] < array[maxChild])**

**{**

**temp = array[root];**

**array[root] = array[maxChild];**

**array[maxChild] = temp;**

**root = maxChild;**

**}**

**else**

**done = 1;**

**}**

**}**

**void heapSort(int array[], int array\_size)**

**{**

**int i, temp;**

**for (i = (array\_size / 2)-1; i >= 0; i--)**

**siftDown(array, i, array\_size);**

**for (i = array\_size-1; i >= 1; i--)**

**{**

**temp = array[0];**

**array[0] = array[i];**

**array[i] = temp;**

**siftDown(array, 0, i-1);**

**}**

**}**

#### External Sorting Techniques

External sorts are generally carried out when serial files need to be converted to sequential files and there is insufficient space within the main memory of the system to hold all the data at once. As with internal sorting techniques, there are numerous algorithms which are use to perform sorts external to the computer’s main memory. Among the more popular algorithms are:

* Tag Sorts
* Four Tape Sort
* Polyphase Sort
* External Radix Sort
* External Merge.

Of the external sort methods mentioned here the Polyphase sort is more efficient in terms of speed and utilisation of resources. However, it is more complicated and therefore in some situations the other algorithms could be more applicable. In practice these sorting methods are already being supplemented by internal sorts. Thus, a number of records from each tape would

be read into main memory and sorted using an internal sort and then output to the tape rather than one record at a time as was the case initially.