

## Sorting Algorithms

Here is a (too) brief summary of four simple sorting algorithms. The first three take time proportional to the square of the number of elements. The last one is “usually” better than this.

### 1 Selection Sort

*Selection Sort* proceeds as follows. Select the largest element in the array. Move that element to the last position. Repeat, ignoring the last element. And so on. Here is C code:

```
void selectionSort(int A[], int len)
{
    int i,j;
    for(i=0; i<len; i++) {
        int maxPos = 0;
        for(j=1; j<len-i; j++) {
            if( A[j] > A[maxPos] )
                maxPos = j;
        }
        int temp = A[maxPos];
        A[maxPos]=A[len-1-i];
        A[len-1-i]=temp;
    }
}
```

### 2 Bubble Sort

Bubble Sort is an example of an *exchange sort*. An exchange sort means that the sort repeatedly compares pairs of elements and swaps them if needed. Bubble Sort is organized as a series of passes. In each pass, you compare the 1st and 2nd elements and swap if out of order; then the 2nd and 3rd elements; and so on. At the end of the first pass, the largest element is guaranteed to be at the end of the array. (Why?) This sort is implemented in the lab (and minor improvements are discussed there).

### 3 Insertion Sort

*Insertion Sort* proceeds as follows: Add elements one at a time from the right, maintaining the initial portion of the array sorted at each pass. The running time is

fast if the list is already sorted!

```
void insertionSort(int A[], int len)
{
    int i;
    for(i=0; i<len; i++) {
        int inserting = A[i];
        int j=i;
        while(j>0 && inserting<A[j-1]) {
            A[j]=A[j-1];
            j--;
        }
        A[j]= inserting;
    }
}
```

## 4 Shell Sort

The problem with a sort like bubble sort or insertion sort is that a small element can start out far from where it should be, and it moves only one step closer each pass. So a natural idea is to allow elements to make bigger moves early on.

There are several variations of Shell Sort. The general approach is:

*Choose a decreasing sequence of numbers  $h_1, h_2, h_3, \dots$ . In pass  $i$ , do insertion sort on the array split into subarrays containing every  $h_i^{\text{th}}$  element.*

A simple choice of sequence is  $h_1 = n/2$ ,  $h_2 = n/4$ ,  $h_3 = n/8$ , and so on, where  $n$  is the length of the array. The last  $h_i$  is always 1; this means that the last pass is just insertion sort—except that the earlier passes have made the array “nearly sorted”, and so the insertion sort entails much less movement of the elements.

Here is a picture for sorting an array with  $n = 8$ :

