**Selection sort**

### **Proposition.**Selection sort uses ~N2/2 compares and N exchanges to sort an array of length N.

**Insertion sort**

**Proposition.** For randomly ordered arrays of length N with with distinct keys, insertion sort uses ~N2/4 compares and ~N2/4 exchanges on the average. The worst case is ~ N2/2 compares and ~ N2/2 exchanges and the best case is N-1 compares and 0 exchanges.

Insertion sort works well for certain types of nonrandom arrays that often arise in practice, even if they are huge. An *inversion* is a pair of keys that are out of order in the array.

**Shellsort**

### **Property.**The number of compares used by shellsort with the increments 1, 4, 13, 40, 121, 364, ... is bounded by a small multiple of N times the number of increments used.

**Abstract in-place merge**

Mergesort guarantees to sort an array of N items in time proportional to N log N, no matter what the input. Its prime disadvantage is that it uses extra space proportional to N.

**Top-down mergesort**

### **Proposition.**Top-down mergesort uses between 1/2 N lg N and N lg N compares and at most 6 N lg N array accesses to sort any array of length N.

**Bottom-up mergesort**

**Proposition.**Bottom-up mergesort uses between 1/2 N lg N and N lg N compares and at most 6 N lg N array accesses to sort any array of length N.

**Quicksort**

It works by *partitioning* an array into two parts, then sorting the parts independently.

### **Proposition.**Quicksort uses ~2 N ln N compares (and one-sixth that many exchanges) on the average to sort an array of length N with distinct keys.

## **Heapsort**

 We can use any priority queue to develop a sorting method. We insert all the keys to be sorted into a minimum-oriented priority queue, then repeatedly use*remove the minimum* to remove them all in order. When using a heap for the priority queue, we obtain *heapsort*.

**Proposition.** Heapsort users fewer than 2N lg N compare and exchanges to sort N items.