We have already seen one sorting algorithm: Heap Sort. This has running time $O(n \log n)$. Below are four more comparison-based sorts; that is, they only compare entries. (An example of an alternative sort is \textit{radix} sort of integers, which directly uses the bit pattern of the elements.)

### 19.1 Insertion Sort

Insertion Sort is the algorithm that:

\textit{adds elements one at a time, maintaining a sorted list at each stage.}

Say the input is an array. Then the natural implementation is such that the sorted portion is on the left and the yet-to-be-examined elements are on the right.

In the worst case, the running time of Insertion Sort is $O(n^2)$; there are $n$ additions each taking $O(n)$ time. For example, this running time is achieved if the list starts in exactly reverse order. On the other hand, if the list is already sorted, then the sort takes $O(n)$ time. (Why?)

Insertion Sort is an example of an \textit{in situ} sort; it does not need extra temporary storage for the data. It is also an example of a \textit{stable sort}: if there are duplicate values, then these values remain in the same relative order.

### 19.2 Shell Sort

Shell Sort was invented by D.L. Shell. The general version is:

0. Let $h_1, h_2, \ldots, h_k = 1$ be a decreasing sequence of integers.

1. For $i = 1, \ldots, k$: do Insertion Sort on each of the $h_i$ subarrays created by splitting the array into every $h_i^{th}$ element.

Since in phase $k$ we end with a single Insertion Sort, the process is guaranteed to sort.

Why then the earlier phases? Well, in those phases, elements can move farther in one step. Thus, there is a potential speed up. The most natural choice of sequence is $h_i = n/2^i$. On average this choice does well; but it is possible to concoct data where this still takes $O(n^2)$ time. Nevertheless, there are choices of the $h_i$ that guarantee Shell Sort takes better that $O(n^2)$ time.

### 19.3 Merge Sort

Merge Sort was designed for computers with external tape storage. It is a recursive divide-and-conquer algorithm:
1. Arbitrarily split the data
2. Call MergeSort on each half
3. Merge the two sorted halves

The only step that actually does anything is the merging. The question is: how to merge two sorted lists to form one sorted list. The algorithm is:

*repeatedly: compare the two elements at the tops of both lists, removing the smaller.*

The running time of Merge Sort is $O(n \log n)$. The reason for this is that there are $\log_2 n$ levels of the recursion. At each level, the total work is linear, since the merge takes time proportional to the number of elements.

Note that a disadvantage of Merge Sort is that extra space is needed (this is not an *in situ* sort). However, an advantage is that sequential access to the data suffices.

### 19.4 QuickSort

A famous recursive divide-and-conquer algorithm is QuickSort.

1. Pick a pivot
2. Partition the array into those elements smaller and those elements bigger than the pivot
3. Call QuickSort on each piece

The most obvious method to picking a pivot is just to take the first element. This turns out to be a very bad choice if, for example, the data is already sorted. Ideally one wants a pivot that splits the data into two like-sized pieces. A common method to pick a pivot is called *middle-of-three*: look at the three elements at the start, middle and end of the array, and use the median value of these three. The “average” running time of QuickSort is $O(n \log n)$. But one can concoct data where QuickSort takes $O(n^2)$ time.

There is a standard implementation. Assume the pivot is in the first position. One creates two “pointers” initialized to the start and end of the array. The pivot is removed to create a hole. The pointers move towards each other, one always pointing to the hole. This is done such that: the elements before the first pointer are smaller than the pivot and the elements after the second are larger than the pivot, while the elements between the pointers have not been examined. When the pointers meet, the hole is refilled with the pivot, and the recursive calls begin.
19.5 Lower Bound for Sorting

Any comparison-based sorting algorithm has running time at least $O(n \log n)$.

Here is the idea behind this lower bound. First we claim that there are essentially $n!$ possible answers to the question: what does the sorted list look like. One way to see this, is that sorting entails determining the rank (1 to $n$) of every element. And there are $n!$ possibilities for the list of ranks.

Now, each operation (such as a comparison) reduces the number of possibilities by at best a factor of 2. So we need at least $\log_2(n!)$ steps to guarantee having narrowed down the list to one possibility. (The code can be thought of as a binary decision tree.) A mathematical fact (using Stirling’s formula) is that $\log_2(n!)$ is $O(n \log n)$.

Sample Code

Here is template code for Insertion Sort. We also introduce the idea of a comparator, where the user can specify how the elements are to be compared.

```cpp
Sorting.cpp
```