Let’s look at the most common algorithms for sorting. You might ask why we need to talk about sorting algorithms at all, given that sorting algorithms are built into Java and other common languages these days. One reason is that you will probably have to use an environment in which sorting is not so available. It is also useful to understand the tradeoffs between different sorting algorithms.

1 Insertion sort

Insertion is a simple algorithm that is very fast for small arrays. Intuitively, insertion sort puts each element into the place it belongs. The algorithm splits the array into a sorted part and an unsorted part. Each element is turn is inserted into the sorted part, causing it to grow. Eventually all elements have been inserted into the sorted part and the algorithm terminates.

```java
/** Sort a[l..r-1]. */
void sort(int[] a) {
    for (int i = 1; i < a.length; i++) {
        int k = a[i];
        int j = i;
        for (; j > 0 && a[j-1] > k; j--)
            a[j] = a[j-1];
        a[j] = k;
    }
}
```

The loop invariant for the outer loop is as depicted in Figure 1. The invariant is satisfied when $i = 1$ and each loop iteration ensures that the value $i$ currently points to, $k$, is inserted in the right place.

The invariant for the inner loop is also illustrated in Figure 1. The index $j$ points to an array location such that everything to the left of $j$ (region A) is less than everything to the right (region B). Further, everything in region B is greater than the value to be inserted, $k$. When the loop terminates, the top element in A is less than or equal to the $k$, so $k$ can be placed in the element marked “?”. Figuring out loop invariants helps us write code like this that is efficient and correct.

The running time of insertion sort is best when the array is already sorted. In this case the inner loop stops immediately on each outer iteration, so the total work done per outer iteration is constant. Therefore the total work done by the algorithm is linear in the array size, or $O(n)$.

The worst case for the algorithm is when the array is sorted in the reverse order. In that case the loop on $j$ goes all the way to 0 on each outer iteration. The first iteration does one copy, the second two copies, and so on, so the total work is $\sum\{1, \ldots, n-1\} = n(n-1)/2$. This function is $O(n^2)$, since $O(n^2 - n) = O(n^2)$.

![Figure 1: Outer and inner loop invariants](image)
Note that in general, we can drop lower-order terms from polynomials when determining asymptotic complexity. For example, in this case \((n^2 - n)/n\) limits to a constant (1) as \(n\) becomes large. Therefore the two functions in the ratio have the same asymptotic complexity.

Insertion sort has one other nice property: it is a stable sort, meaning that if given an array containing elements that are equal to each other, those elements are kept in the same order relative to each other.

## 2 Selection sort

Selection sort is another algorithm commonly used by humans. Intuitively, it tries to find the right element to put in each location of the final array.

```java
for (int i = 0; i < n; i++) {
    // find smallest element in subarray a[i..n-1]
    // swap it with a[i]
}
```

Because each loop iteration must in turn iterate over the rest of the array to find the smallest element, the best-case performance of this algorithm is the same as the worst-case performance: \(O(n^2)\).

## 3 Merge sort

More efficient sorting algorithms use the divide-and-conquer strategy. They break the array into smaller subarrays and recursively sort them. Mergesort is one such algorithm. Given an array to sort, it finds the middle of the array and then recursively sorts the left half and the right half of the array. Then it merges the resulting arrays. A temporary array \(tmp\) is provided to give space for merging work:

```java
/** Sort a[l..r-1]. Modifies tmp.
    Requires: l < r, and tmp is an array at least as long as a.
*/
void sort(int[] a, int l, int r, int[] tmp) {
    if (l == r-1) return; // already sorted
    int m = (l+r)/2;
    sort(a, l, m, tmp);
    sort(a, m, r, tmp);
    merge(a, l, m, r, tmp)
}
```

The real work is done in `merge`, which takes time linear in the total number of elements to be merged: \(O(r-l)\).

```java
/** Place a[l..r-1] into sorted order.
   * Requires: l < m < r, and a[l..m-1] and a[m..r-1] are both in sorted order.
   * Performance: O(r-l)
   */
void merge(int[] a, int l, int m, int r, int[] b) {
    int i = l;
    int j = m;
    int k = l;
    while (i < m && j < r)
        tmp[k++] = (a[i] < a[j]) ? a[i++] : a[j++];
    Arrays.arraycopy(a, i, tmp, k, m-i);
    Arrays.arraycopy(a, j, tmp, k, r-j);
    Arrays.arraycopy(y, l, x, l, r-l);
}
The run time of this algorithm is always $O(n \lg n)$, which is big improvement on $O(n^2)$. For example, if sorting a million elements, the speedup, ignoring constant factors, is $1,000,000 / \lg 1,000,000 \approx 50,000$. The speedup probably won’t be quite that great when comparing to insertion sort because of constant factors.

To see why it is $n \lg n$, think about the whole sequence of recursive calls shown in Figure 2. Each layer of recursive calls takes total merge time proportional to $n$, and there are $\lg n$ recursive calls. The total time spent in the algorithm is therefore $O(n \cdot \lg n)$.

Merge sort, like insertion sort, is a stable sort. This is a major reason when merge sort is commonly used. Another is that its run time is predictable.

Merge sort is not as fast as the quicksort algorithm that we will see next because it does extra copying into the temporary array. It is possible to do a linear-time in-place merge, but this is quite tricky and turns out to be even slower than copying into a separate array. Some speedup can be achieved by avoiding the final copy from `tmp` back into `a`. This can be accomplished by interchanging the roles of `tmp` and `a` on each recursive call.

Another trick that is used to speed up mergesort is to use insertion sort when the subarrays get small enough. For very small arrays insertion sort will be faster, because $k_1 n^2$ is smaller than $k_2 n \lg n$ when $n$ and $k_1$ are small enough!

4 Quicksort

Quicksort is another divide-and-conquer sorting algorithm. It avoids the work of merging by partitioning the array elements before recursively sorting. The algorithm chooses a pivot value $p$ and 1 elements at least as large as $p$ and the left half contains elements no larger than $p$. These two subarrays can then be sorted recursively and the algorithm is done.
/** Sort a[l..r-] */
void qsort(int[] a, int l, int r) {
    if (l == r-1) return // base case: already sorted
    int p = a[l]; // probably want to swap a different element with a[l] first.
    // partition elements using p, obtaining partition point k
    qsort(a, l, k);
    qsort(a, k, r);
}

One thing we notice is that the choice of pivot matters. If the pivot value is the largest or smallest element in the array, the subarrays have lengths 1 and n − 1. If this happens on every recursion—which it easily can if the array is sorted to begin with—quicksort will take $O(n^2)$ time. One solution is to choose the pivot randomly from among the elements of the array. With this choice quicksort has expected run time $O(n \log n)$, by reasoning similar to that for mergesort. A second commonly used heuristic is to choose the median of the first, the last, and the middle element of the array. This heuristic makes quicksort perform well on arrays that are mostly sorted.

Now, how to partition elements efficiently? We want the array to end up looking like this:

```
≤ p
k
≥ p
```

The idea is to start two pointers $i$ and $j$ from opposite ends of the array. They sweep in toward the middle swapping elements to achieve the final partitioned state shown above. Initially the array looks like this:

```
p
?
i
j
```

We need an invariant that starts out describing the initial state and ends up describing the final state. As the following diagram suggests, the invariant says that all elements strictly to the left of $i$ are at most $p$, and all elements strictly to the right of $j$ are no less than $p$.

```
≤ p
?  ≥ p
i  j
```

There is a second part to the invariant that is not depicted, but that is needed to make sure that $i$ and $j$ do not fall off the end of the array: the array elements to the right of $i$, inclusive, must include at least one element that is at least $p$, and the elements to the left of $j$, inclusive, must include at least one element that is at most $p$. Clearly this condition is satisfied in the initial state.

The partitioning code then looks very simple:

```c
while (true) {
    while (a[j] > p) j--;
    while (a[i] < p) i++;
    if (i == j) break;
    swap a[i] ↔ a[j]
    i++; j--;
}
k = j+1;
```
Notice that in the inner loops we do not need to test whether \( i \) and \( j \) are going out of bounds, thanks to the second part of the invariant. This is an example of how getting the invariant just right can enable more efficient code.

An example of partitioning will probably help understand what is going on. We start out with the following array, with \( p=5 \):

\[
\begin{array}{cccc}
5 & 2 & 7 & 9 \\
i & j
\end{array}
\]

In the first iteration, neither \( i \) nor \( j \) move, so we swap their elements and bump both inward:

\[
\begin{array}{cccc}
3 & 2 & 7 & 9 \\
i & j
\end{array}
\]

In the second iteration, \( j \) moves all the way down to 2, and \( i \) moves up to 7:

\[
\begin{array}{cccc}
3 & 2 & 7 & 9 \\
j & i
\end{array}
\]

Since \( i \geq j \), the loop halts and \( k \) points to \( j+1 = 7 \). The two subarrays to be recursively sorted are \((3,2)\) and \((7,9,5)\).

Quicksort is an excellent sorting algorithm for many applications. However, one downside is that it is not a stable sort. As with mergesort, it may make sense to switch to insertion sort for sufficiently small subarrays.