This handout presents three different sorting routines: selection sort, merge sort, and quick sort. Each of these routines takes four arguments: an array of items to sort; the low and high indices in the array between which lie the elements we wish to sort; and a comparator that takes two items and returns `precedes`, `equal_to` or `follows` according to the order relation between its two arguments. We'll assume that `array_item_type` has been defined to be the type of the array elements being sorted, and `array_type` has been defined to be an array of `array_item_type`s. First, here are the `comparison_result` type, the `comparator_type` type, and the the swap routine that are used by all of the sort routines:

```
typedef enum {
  precedes,
  equal_to,
  follows
} comparison_result;
```

```
typedef comparison_result (*comparator_type)(array_item_type, array_item_type);
```

```
void
swap(array_item_type *a, array_item_type *b)
{
  array_item_type temp;

  temp = *a;
  *a = *b;
  *b = temp;
}
```

Here is `selection_sort`. It works by repeatedly finding the next item in the sequence and putting that item in its proper place.

```
void
selection_sort(array_type items,
               int low,
               int high,
               comparator_type comparator)
{
  int pos_to_fill;

  for (pos_to_fill = low; pos_to_fill < high; pos_to_fill++) {
    int pos_of_lowest_remaining;
    int i;

    /* find the position of the lowest remaining value. */
    pos_of_lowest_remaining = pos_to_fill;
    for (i = pos_to_fill + 1; i <= high; i++)
      if (comparator(items[i], items[pos_of_lowest_remaining]) == precedes)
        pos_of_lowest_remaining = i;
    /* move the lowest remaining value to its proper place. */
    swap(&items[pos_to_fill], &items[pos_of_lowest_remaining]);
  }
}
```

`Merge_sort` relies on the `merge` function to merge together two sorted lists. `Merge_sort` gets its efficiency from the fact that at most one comparison needs to be made for each number being merged.

```
void
merge(array_type arr,
      int low1, int high1,
      int low2, int high2,
      comparator_type comparator)
{
  array_type temp;
  int temp_index = low1;
  int index1 = low1;
  int index2 = low2;
  int i;

  /* Make sure that the two areas are contiguous. */
  assert(high1 + 1 == low2);
  /* While there are elements in both halves, copy the lowest one */
  while (index1 <= high1 && index2 <= high2) {
    if (comparator(arr[index1], arr[index2]) == precedes)
      temp[temp_index++] = arr[index1++];
    else temp[temp_index++] = arr[index2++];
  }
  /* Copy all the remaining elements.  Only one of these loops will do any copying. */
  while (index1 <= high1)
    temp[temp_index++] = arr[index1++];
  while (index2 <= high2)
    temp[temp_index++] = arr[index2++];
  /* Copy the now-sorted elements back into the original array */
  for (i = low1; i <= high2; i++)
    arr[i] = temp[i];
}
```

`Merge_sort` itself is quite simple—divide the array into two pieces, sort each piece, then merge the two pieces together.

```
void
merge_sort(array_type arr,
   int low,
   int high,
   comparator_type comparator)
{
  int mid;

  /* Make sure that there are at least two elements to sort.  If
     not, then the items between low and high are already sorted. */
  if (low < high) {
    mid = (low + high) / 2;
    merge_sort(arr, low, mid, comparator);
    merge_sort(arr, mid + 1, high, comparator);
    merge(arr, low, mid, mid + 1, high, comparator);
  }
}
```

Just like `merge_sort`, `quick_sort` splits the array into two pieces. However, `quick_sort` takes the additional step of placing all the low values in the left piece and all the high values in the right piece. Consequently, no merge step is needed after the two halves have been sorted.

The `split` function is responsible for dividing the array into two pieces. It starts an index at each end of the array and moves them inward. Whenever two values that are in the wrong halves are found, they are swapped. When the two indices meet, the split is complete.

```
  /* Split places all elements that precede its guess of the median
     at the left of arr, and all elements that do not precede the
     median_guess at the right of arr.  Split returns the index of the
     last element in the first half. */
int
split(array_type arr,
      int low,
      int high,
      comparator_type comparator)
{
  array_item_type median_guess;

  /* Select arr[low] as the median_guess, and split the array into items
       that precede it and items that do not.  A better approach would be to
       select an element at random to serve as the median_guess. */
  median_guess = arr[low];
  for (;;) {
    while (comparator(arr[high], median_guess) == follows)
      high--;
    while (comparator(arr[low], median_guess) == precedes)
      low++;
    if (low < high)
      swap(&arr[low++], &arr[high--]);
    else return(high);
  }
}
```

`Quick_sort` itself looks similar to `merge_sort`, except for the use of `split` to select the midpoint, and the deletion of the final merge.

```
void
quick_sort(array_type arr,
   int low,
   int high,
   comparison_result (*comp)(array_item_type, array_item_type))

{
  /* Make sure that there are at least two elements to sort.  If
     not, then the items between low and high are already sorted. */
  if (low < high) {
    int mid;

    mid = split(arr, low, high, comp);
    quick_sort(arr, low, mid, comp);
    quick_sort(arr, mid + 1, high, comp);
  }
}
```

These three sorting routines, along with a silly driver program, are available in the file `~mayfield/sort.c`.