1. Useful functions
   1. // Swaps the integers pointed to by a and b.
   2. void swap(int \*a, int \*b) {
      1. int temp = \*a;
      2. \*a = \*b;
      3. \*b = temp;
   3. }
   4. //The contents of A from index 0 to index n-1 are printed to the screen separated by spaces followed by a newline.
   5. void printArray(int A[], int n) {
      1. int i;
      2. // Go through each array element one by one.
      3. for (i = 0; i<n; i++)
         1. printf("%4d", A[i]);
      4. printf("\n");
   6. }
2. O(*n*2) Sorts
   1. Bubble Sort
      1. Logic
         1. One iteration
            1. “King of the hill” swapping beginning with largest number
            2. At the end of the battle, the largest reaches the end of the array as winner.
         2. Now, sort everything behind the winner (the remaining *n* – 1 terms) recursively.
            1. Generally sorting from smallest to largest
      2. Runtime
         1. 1st iteration: *n* steps
         2. 2nd iteration: *n* – 1 steps
         3. *n*th iteration: 1 step
         4. Runtime =
      3. Code
         1. void bubbleSort(int A[], int n) {
            1. int i, j;
            2. // Loop through each element to insert.
            3. for (i=n-2; i>=0; i--) {

for (j=0; j<=i; j++)

if (A[j] > A[j+1])

swap(&A[j], &A[j+1]);

if (0) {

printf("Here is the array now: ");

printArray(A, n);

}

* + - * 1. }
      1. }
    1. Exercise: write Bubble Sort recursively.
       1. Code
          1. void bubbleSort(int A[], int n) {

int i, j;

//Run One Pass

if (n == 1)

return;

bubblesort(Array,n-1);

* + - * 1. }
      1. Runtime: a common recurrence relation is T(*n*) = f(*n*) + T(*n* – 1) =
         1. T(*n*) = *n* + T(*n* – 1), … T(1)
         2. *= n* + (*n* – 1) + (*n* – 2)
         3. =
         4. =
    1. Optimization
       1. Add Boolean variable to mark if something has been swapped or not.
  1. Insertion Sort
     1. Logic
        1. For each item from *i* to *n* – 1, insert item i into its correct location in the already-sorted array from index [0, i – 1].
           1. Use a while loop
     2. Code
        1. void insertionSort(int A[], int n) {
           1. int i, j;
           2. // Loop through each element to insert.
           3. for (i = 1; i < n; i++) {

j = i;

//Continue swapping the element until it hits the correct location.

while (j > 0 && A[j] < A[j-1]) {

swap(A+j, A+j-1);

j--;

}

if (0) {

printf("Here is the array now: ");

printArray(A, n);

}

* + - * 1. }
      1. }
    1. Example: 3, 7, 2, 1, 5
       1. 3, 7, 2, 1, 5, since 7 > 3, so the list 3, 7 is sorted.
       2. 3, 2, 7, 1, 5, since 2 < 7
       3. 2, 3, 7, 1, 5, since 2 < 3, so the list 2, 3, 7 is sorted.
       4. 2, 3, 1, 7, 5, since 1 < 7,
       5. 2, 1, 3, 7, 5, since 1 < 3,
       6. 1, 2, 3, 7, 5, since 1 < 2, so the list 1, 2, 3, 7 is sorted.
       7. 1, 2, 3, 5, 7, since 5 < 7 and now we can stop since 5 > 3.
  1. Selection Sort
     1. Logic
        1. Find index from max item from index 0 to i.
        2. Swap max index with i.
        3. Determine the smallest element in the rest of the array to the right of the ith element.
        4. Swap the current ith element with the element identified in step 1.
     2. Runtime
        1. O(*n*2)
        2. Cases
           1. Best case: already sorted O(*n*)
           2. Worst case: reverse list O(*n*2)
           3. Average case: O(0.5*n*2) 🡪 O(*n*2)
     3. Code
        1. void selectionSort(int A[], int n) {
           1. int cur, j, smallest;
           2. // Loop through each index of the array. At each loop iteration we will be placing the smallest unplaced item left in this location of the array.
           3. for (cur = 0; cur <n; cur++) {

// At first, the smallest unplaced element is stored in the index cur.

smallest = cur;

// Look through the other indices to find a value SMALLER than [cur]. If we find one, update WHERE it is located.

for (j=cur+1; j<n; j++)

// We found a smaller element!

if (A[j] < A[smallest])

smallest = j;

// Now, swap the smallest element left, list[smallest] into its correctly sorted location, list[cur].

swap(&A[cur], &A[smallest]);

if (0) {

printf("Here is the array now: ");

printArray(A, n);

}

* + - * 1. }
        2. printf("\n");
      1. }

1. Merge Sort
   1. Problem
      1. 2 sorted lists with variable number of items
         1. List 1 (0 to i): 2, 4, 5, 9, 15, 18, 27, 33
         2. List 2 (0 to j): 3, 6, 10, 11, 12, 22
      2. Create a sorted List 3
         1. List 3 = {List 1, List 2}
   2. Logic
      1. Compare first items
         1. Whichever is smaller, copy to list 3
         2. Increment index i or j for list 1 and list 2 respectively
      2. Once the smaller list runs out of elements
         1. Copy the last elements of the larger list in order to List 3.
   3. Runtime
      1. O(*n* + *m*)
   4. Code
      1. void MergesSort(int values[], int start, int middle, int end) {
         1. int \*temp, i, length, count1, count2, mc;
         2. // Allocate the proper amount of space for our auxiliary array.
         3. length = end - start + 1;
         4. temp = (int\*)calloc(length, sizeof(int));
         5. // These will be our indexes into our two sorted lists.
         6. count1 = start;
         7. count2 = middle;
         9. // Keeps track of our index into our auxiliary array.
         10. mc = 0;
         11. // Here we copy values into our auxiliary array, so long as there are numbers from both lists to copy.
         12. while ((count1 < middle) || (count2 <= end)) {
             1. // Next value to copy comes from list one - make sure list one isn't exhausted yet. Also make sure we don't access index ount2 if we aren't supposed to.
             2. if (count2 > end || (count1 < middle && values[count1] < values[count2])) {

temp[mc] = values[count1];

count1++;

mc++;

* + - * 1. }
        3. // We copy the next value from list two.
        4. else {

temp[mc] = values[count2];

count2++;

mc++;

* + - * 1. }
      1. }
      2. //Copy back all of our values into the original array.
      3. for (i=start; i<=end; i++)
         1. values[i] = temp[i - start];
      4. // Don't need this space any more!
      5. free(temp);
    1. }
  1. Recursive Version
     1. void RecursiveMergeSort(int values[], int start, int end) {
        1. int mid;
        3. // Check if our sorting range is more than one element.
        4. if (start < end) {
           1. mid = (start+end)/2;
           3. // Sort the first half of the values.
           4. MergeSort(values, start, mid);
           6. // Sort the last half of the values.
           7. MergeSort(values, mid+1, end);
           8. // Put it all together.
           9. Merge(values, start, mid+1, end);
        5. }
     2. }