Arrays

A very common application of arrays is to sort a collection of numbers (or chars) into an ordered ascending or descending sequence.

Imagine a random sequence of integers: 3 6 2 9 4 7 1. We want to sort them into an ascending sequence (lowest number first, highest number last).

Our human brains are actually pretty good at doing this sort of thing, and we can probably arrange the numbers correctly by inspection — as long as the list of number isn't too big. But imagine doing this like a computer, which is stupid, but fast. The computer can compare only two items at a time, but it can do a lot of comparisons. Imagine going through the list one pair of items at a time, comparing them, and then swapping them in the list if the first is bigger than second. As we go through the list, the sequence would go like this:

Compare 3 and 6. Is 3 > 6? No. So make no change. 3 6 2 9 4 7 1

Compare 6 and 2. Is 6 > 2? Yes. So swap them. 3 2 6 9 4 7 1

Compare 6 and 9. Is 6 > 9? No. So make no change. 3 2 6 9 4 7 1

Compare 9 and 4. Is 9 > 4? Yes. So swap them. 3 6 2 4 9 7 1

Compare 9 and 7. Is 9 > 7? Yes. So swap them. 3 2 6 4 7 9 1

Compare 9 and 1. Is 9 > 1? Yes. So swap them. 3 2 6 4 7 1 9

So with 6 comparisons, we have gone through the list once. It more ordered than the original, but it still not completely ordered. So we should go through it again. And again. And again. How many times? For this list, it will take 6 times through to guarantee to that we have made every possible comparison needed to order the list. Or more generally, if there are n items, we need n – 1 comparisons within the list, and then we need to go through that entire process n – 1 times.

The sequence of numbers at each step in this "two-loop" process is shown on the next page.

3 6 2 9 4 7 1

1-1: 3 6 2 9 4 7 1

1-2: 3 2 6 9 4 7 1

1-3: 3 2 6 9 4 7 1

1-4: 3 2 6 4 9 7 1

1-5: 3 2 6 4 7 9 1

1-6: 3 2 6 4 7 1 9

2-1: 2 3 6 4 7 1 9

2-2: 2 3 6 4 7 1 9

2-3: 2 3 4 6 7 1 9

2-4: 2 3 4 6 7 1 9

2-5: 2 3 4 6 1 7 9

2-6: 2 3 4 6 1 7 9

2 3 4 6 1 7 9

3-1: 2 3 4 6 1 7 9

3-2: 2 3 4 6 1 7 9

3-3: 2 3 4 6 1 7 9

3-4: 2 3 4 1 6 7 9

3-5: 2 3 4 1 6 7 9

3-6: 2 3 4 1 6 7 9

4-1: 2 3 4 1 6 7 9

4-2: 2 3 4 1 6 7 9

4-3: 2 3 1 4 6 7 9

4-4: 2 3 1 4 6 7 9

4-5: 2 3 1 4 6 7 9

4-6: 2 3 1 4 6 7 9

2 3 1 4 6 7 9

5-1: 2 3 1 4 6 7 9

5-2: 2 1 3 4 6 7 9

5-3: 2 1 3 4 6 7 9

5-4: 2 1 3 4 6 7 9

5-5: 2 1 3 4 6 7 9

5-6: 2 1 3 4 6 7 9

6-1: 1 2 3 4 6 7 9

6-2: 1 2 3 4 6 7 9

6-3: 1 2 3 4 6 7 9

6-4: 1 2 3 4 6 7 9

6-5: 1 2 3 4 6 7 9

6-6: 1 2 3 4 6 7 9

The process required (n – 1)*(n – 1) comparisons, and a few at the end were pointless, but it got the job done.

This is called a “bubble sort” (bubbles rise to the top) and is easy to implement with an array and some nested loops.

The sequence of steps is to (1) assign the n values to an array with n elements. (2) Set up an inner loop that goes through the array one item at a time, comparing the item to its neighbor, and swapping them if appropriate. (3) Then an outer loop repeats this “compare and swap” process n-1 times.

```c
//  arrays and sorting
//
//  Created by Gary Tuttle on 9/18/16.
//  Copyright © 2016 Gary Tuttle. All rights reserved.
//

#include <stdio.h>

int main( void) {

    const int NUMBER_OF_ITEMS = 7;    //the number of items to be sorted

    int i, j;                          //counters for the two loops

    int list[NUMBER_OF_ITEMS];        //the array for our items

    int swap;                         //a dummy variable for swapping values


    for(i = 0; i < NUMBER_OF_ITEMS; i++){
        printf( "Enter item %d: ", i);
        scanf("%d", &list[i]);
    }

    printf( "\n\nThe list before ordering: ");

    for(i = 0; i < NUMBER_OF_ITEMS; i++)
        printf( " %d", list[i]);


    for(i = 0; i < NUMBER_OF_ITEMS - 1; i++){

        for( j = 0; j < NUMBER_OF_ITEMS - 1; j++){

            if( list[j] > list[j + 1]){

                swap = list[j];
                list[j] = list[j + 1];
                list[j + 1] = swap;
            }
        }
    }

    printf( "\n\nThe list after ordering: ");

    for(i = 0; i < NUMBER_OF_ITEMS; i++)
            printf( " %d", list[i]);


    printf( "\n\n");
    return 0;
}
```

```
Enter item 0: 9
Enter item 1: 12
Enter item 2: 4
Enter item 3: 16
Enter item 4: 8
Enter item 5: 1
Enter item 6: 11


The list before ordering:  9 12 4 16 8 1 11

The list after ordering:  1 4 8 9 11 12 16

Program ended with exit code: 0
```

```c
//  arrays and sorting
//
//  Created by Gary Tuttle on 9/18/16.
//  Copyright © 2016 Gary Tuttle. All rights reserved.
//

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main( void) {

    const int NUMBER_OF_ITEMS = 100;   //the number of items to be sorted
    int i, j;                          //counters for the two loops
    int list[NUMBER_OF_ITEMS];         //the array for our items
    int swap;                          //a dummy variable for swapping values

    srand( (int)(time(0)));

    for(i = 0; i < NUMBER_OF_ITEMS; i++)    //Use the random number thing to generate list values.
        list[i] = rand()%500;               //The numbers are between 0 and 499.

    printf( "\n\nThe list before ordering: \n\n");  //print it before sorting
    for( i = 0; i < 5; i++){
        for(j = 0; j < 20; j++){
            printf("%d ", list[20 * i + j]);
        }
        printf( "\n\n");
    }

    //Put everything in order with the bubble sort

    for(i = 0; i < NUMBER_OF_ITEMS - 1; i++){

        for( j = 0; j < NUMBER_OF_ITEMS - 1; j++){

            if( list[j] > list[j + 1]){

                swap = list[j];
                list[j] = list[j + 1];
                list[j + 1] = swap;
            }
        }
    }

    printf( "\n\nThe list after ordering: \n\n");  //print it after sorting
    for( i = 0; i < 5; i++){
        for(j = 0; j < 20; j++){
            printf("%d ", list[20 * i + j]);
        }
        printf( "\n\n");
    }

    printf( "\n\n");
    return 0;
}
```

```
The list before ordering:

346 18 340 177 397 268 424 381 179 471 120 270 473 5 46 42 59 116 319 53

163 357 350 267 228 27 228 59 276 470 153 53 338 98 404 351 103 153 443 184

3 336 295 325 284 391 289 220 8 96 209 325 160 113 376 37 237 262 375 67

78 381 141 386 308 166 489 277 127 75 306 44 163 66 12 328 35 260 8 477

364 132 373 459 15 108 399 172 55 239 140 339 98 453 396 353 114 170 61 248


The list after ordering:

3 5 8 8 12 15 18 27 35 37 42 44 46 53 53 55 59 59 61 66

67 75 78 96 98 98 103 108 113 114 116 120 127 132 140 141 153 153 160 163

163 166 170 172 177 179 184 209 220 228 228 237 239 248 260 262 267 268 270 276

277 284 289 295 306 308 319 325 325 328 336 338 339 340 346 350 351 353 357 364

373 375 376 381 381 386 391 396 397 399 404 424 443 453 459 470 471 473 477 489


Program ended with exit code: 0
```

Sorting programs can always be made more efficient. The method that any program using to do computations is called the algorithm. Making algorithms more efficient is an important aspect of computer science. We will not delve deeply into algorithm development in 285, but the study of the general theory of algorithms is something you might consider if you are interested in being a better programmer. (ComSci 311.)

One simple way to improve our bubble algorithm is to note that is quite possible that the list is completely sorted before doing the full $(n-1)^2$ iterations. To do this, we can make a check to see if any changes have occurred at a particular iteration. If no further changes were made, then the list is sorted and we can stop.

We will need to add a variable that is set to true if no swaps occur at a particular step. Before iterating on the outer loop, we can check this variable and stop if it is true.

```c
//  arrays and sorting
//  Created by Gary Tuttle on 9/18/16.
//  Copyright © 2016 Gary Tuttle. All rights reserved.
//

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main( void) {

    const int NUMBER_OF_ITEMS = 100;   //the number of items to be sorted
    int i, j;                          //counters for the two loops
    int list[NUMBER_OF_ITEMS];         //the array for our items
    int swap;                          //a dummy variable for swapping values
    int listChanged = 1;               //a "boolean" for checking to see if changes have occurred.
    int iterationCount = 0;            //For fun, let's count the iterations.

    srand( (int)(time(0)));

    for(i = 0; i < NUMBER_OF_ITEMS; i++)    //Use the random number thing to generate list values.
        list[i] = rand()%500;               //The numbers are between 0 and 499.

    printf( "\n\nThe list before ordering: \n\n");  //print it before sorting
    for( i = 0; i < 5; i++){
        for(j = 0; j < 20; j++){
            printf("%d ", list[20 * i + j]);
        }
        printf( "\n\n");
    }

    //Put everything in order with the bubble sort

    for(i = 0; i < NUMBER_OF_ITEMS - 1; i++){

        if( listChanged == 0 )
            break;
        else
            listChanged = 0;

        for( j = 0; j < NUMBER_OF_ITEMS - 1; j++){

            iterationCount++;

            if( list[j] > list[j + 1]){
                swap = list[j];
                list[j] = list[j + 1];
                list[j + 1] = swap;
                listChanged = 1;            //If we are inside this if statement, the list has changed.
            }
        }
    }

    printf( "The number of iterations was %d.\n", iterationCount);
    printf( "The maximum would have been %d.\n\n", (NUMBER_OF_ITEMS-1)*(NUMBER_OF_ITEMS-1));

    printf( "The list after ordering: \n\n");  //print it after sorting
    for( i = 0; i < 5; i++){
        for(j = 0; j < 20; j++){
            printf("%d ", list[20 * i + j]);
        }
        printf( "\n\n");
    }

    printf( "\n\n");
    return 0;
}
```

```
The list before ordering:

53 382 486 449 294 63 99 261 457 305 268 194 350 276 265 288 275 205 367 105

250 141 482 413 246 379 171 335 211 387 177 406 27 449 478 195 306 281 303 116

321 270 344 216 49 324 335 181 163 40 120 301 447 283 54 377 356 330 373 462

79 386 72 496 54 21 173 270 135 208 460 139 342 236 20 457 24 151 54 186

287 193 481 240 217 256 268 256 453 337 318 346 247 493 436 224 419 423 230 258

The number of iterations was 7425.
The maximum would have been 9801.

The list after ordering:

20 21 24 27 40 49 53 54 54 54 63 72 79 99 105 116 120 135 139 141

151 163 171 173 177 181 186 193 194 195 205 208 211 216 217 224 230 236 240 246

247 250 256 256 258 261 265 268 268 270 270 275 276 281 283 287 288 294 301 303

305 306 318 321 324 330 335 335 337 342 344 346 350 356 367 373 377 379 382 386

387 406 413 419 423 436 447 449 449 453 457 457 460 462 478 481 482 486 493 496


Program ended with exit code: 0
```

This is properly ordered, but we were able to skip that last 2376 iterations, about 25% of the maximum possible. Not bad.