Dan Suciu
Report

Compile program with: gcc –Wall –Werror –lm -03 sorting.c sorting_main.c –o proj1
(added the math library to use the pow() function, so must compile with -lm)

Algorithmic descriptions:
Sequence 1 Generation –> Time = O(Nlog(N)^2) Space = O(N)
To generate sequence 1, I decided to control the outer loop with the condition of 2^power being < N. This is because the inner loop controls the power of the 3 term, and when the power of the 3 term is 0, the 2 term is the entire expression. My outer loop increments the power of 2 starting at 0, and the inner loop increments the power of 3 starting at 0. The inner loop runs until the 2^p * 3^q is greater that N. Then, as long as this is not equal to the last value we saved into the sequence array, it is saved into the sequence array and the index is incremented. This saves every permutation 2^p * 3^q that is less than N, which is not in any order. I then use an efficient sorting algorithm qsort to sort my sequence in ascending order.
Sequence 2 Generation –> Time = O(log(N)) Space = O(N)
To generate sequence 2, we begin at a power of 1 and loop until the floor of N/(1.3^power) is less than 1. Inside the loop we change 9s and 10s to 11 and check for any duplicates to be skipped. After both of these checks, we save the floor of N/(1.3^power) into the sequence array, starting at an index of 0. We then increment power and index as long as we save a value into the sequence.
Shell Insertion Sort –> As the problem size grows a power of 10, so do the comparisons, moves, and run times until the size goes from 100,000 to 1,000,000, at which point the comparisons increase by 2 powers of 10.
(10 thousand) Time = 0.0 sec Comparisons = 550711 Moves = 1166240
(100 thousand) Time = 0.05 sec Comparisons = 8605411 Moves = 18089535
(1 million) Time = 0.61 sec Comparisons = 123987151 Moves = 259684562

For the shell insertion sort, first we generate the gap sequence 1 in the same way that we did above. We then sort the gaps using qsort (since sorting the gaps is not what we are testing the run time of here) in ascending order, and then reverse the order to get the gap sequences in descending order. The outer control of our sort stops when we have pulled and used every gap from the sequence. We then enter our inner control, where we start at and index = gap, and increment index while it is less than the size of the array. We store the value at index1 in a temporary variable, index1 moving from gap to size – 1. Our very inner loop then creates a new index, inner, at index1 and terminates when inner is less than or equal to gap, or when the value at inner – gap is greater than the value we have set aside at index1. While this is true we decrement inner by gap and set the value at inner to the value at inner – gap. When we exit the inner loop we set the value at inner to the val set aside from index1.

My shell insertion sort only requires the given space for the array (i.e. not extra space), as it is sorted in place, and the space allocated for the gap sequence. The gap sequence has allocated to it Size * sizeof(long) as I wanted to be safe about how much space I have for the gaps. This array is also freed at the end of the function, eliminating any possible memory leaks. Other than this the only space is for temporary local variables.
Improved Bubble Sort –> As the problem size grows a power of 10, so do the comparisons and moves, just as the shell insertion sort. However, the improved bubble sort starts a power of 10 lower for the number of moves and does not increase by 2 powers of 10 for comparisons at 1,000,000.
(10 thousand) Time = 0.0 sec Comparisons = 306727 Moves = 186408
(100 thousand) Time = 0.02 sec Comparisons = 3966742 Moves = 2438928
(1 million) Time = 0.21 sec Comparisons = 49666762 Moves = 30144183

For our improved bubble sort we generate the gap sequence inside the control loop for the sort, as I was having problems remaining on a gap of 1 at the end of the sort to assure that it is totally sorted unless I didn't use an array of gap sequences. Our outer loop is controlled by and or statement of gap being greater than or equal to 1, and a swap variable being non zero, which lets us know if we have reached a gap of 1, and if we have we also need there to be no recent swaps, meaning the array is not sorted. In the inner loop we compare index of 0 to an index of gap, swap them if need be, and then shift them both over. We will be done when out gap is 1 and we have not made a swap on the last pass through the array.

My improved bubble sort implementation only requires the given space for the array (i.e. not extra space), as it is sorted in place, and enough space for a handful of temporary variables which only have a scope of the sorting function.