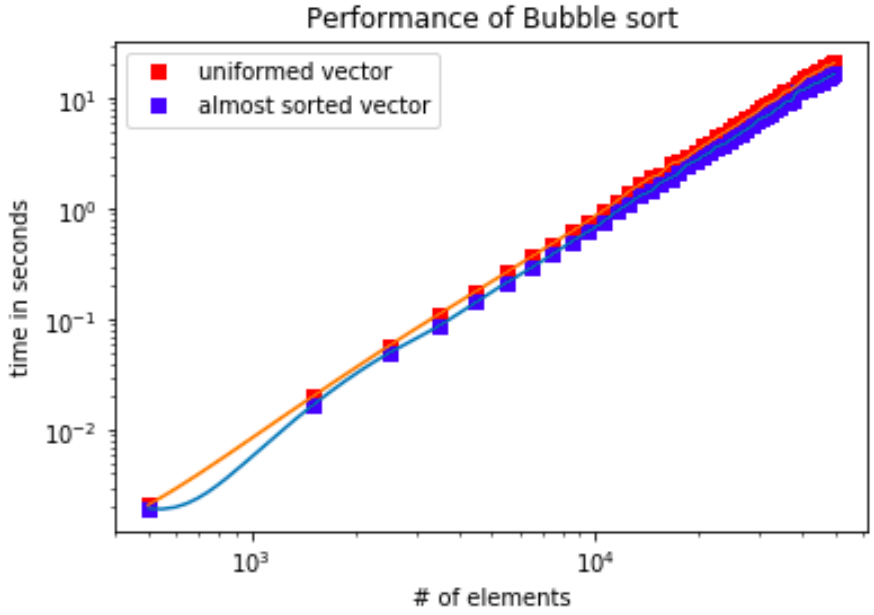
CS 165 Project 1 report

Bubble sort



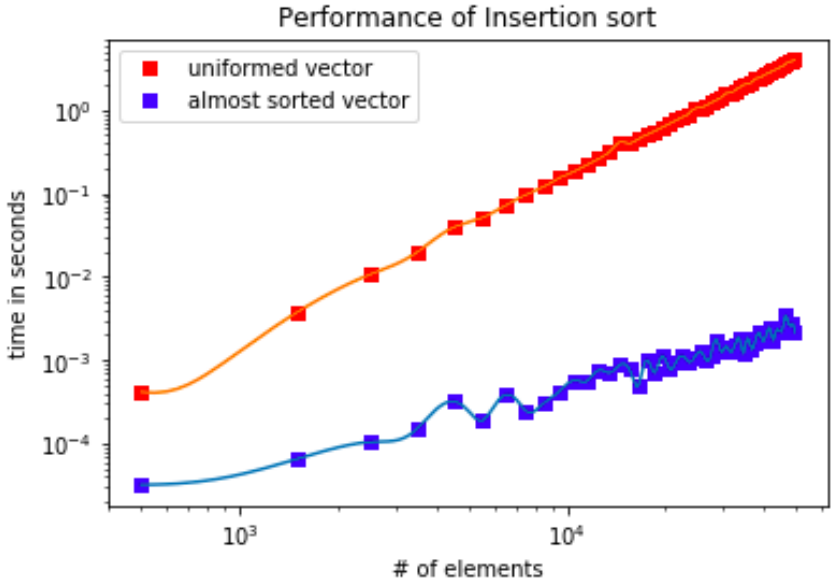
Bubble sort Analysis

Implementation: Repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. If the vector size is 0 or 1,
just return the list. If not, make (n - 1) passes through the list to check adjacent items where n is the length of the list.

Runtime performance: As shown in log-log scale, the performances of bubble sort on Uniformly distributed permutations and Almost-sorted permutations are almost the same, since despite the vector is almost sorted or not, the number of passes is fixed to make sure the vector can be sorted, which means they will have the same iteration time if given same input size. We can tell from the plot that the timing on almost sorted vector is a little bit smaller. The reason for that is almost sorted vectors require less swap operations.

Time complexity: The complexity for bubble sort is O(n^2).

Insertion sort

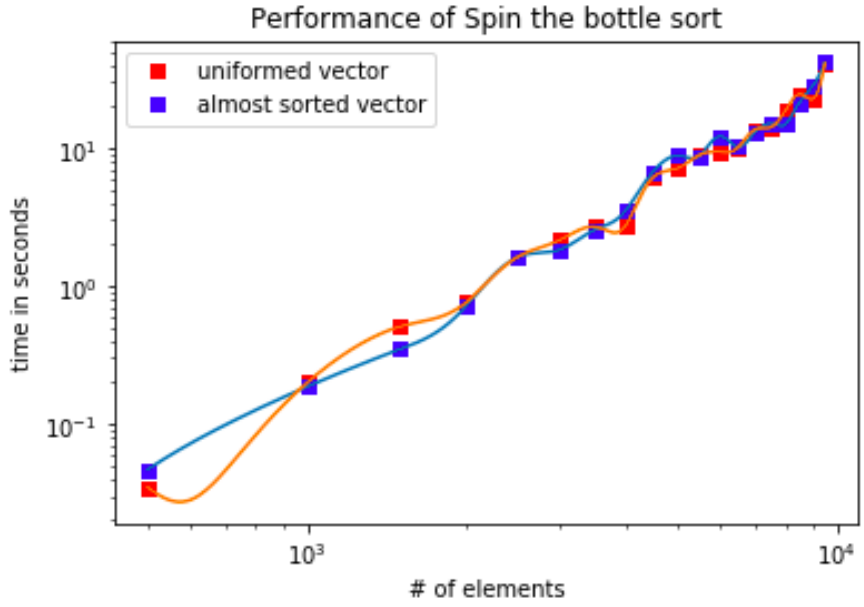


Implementation: Orders a list of values by repetitively inserting a particular value into a sorted subset of the list. If the vector size is 0 or 1, just return the list. If not, Consider the first element to be sorted and the rest to be unsorted. Compare the first element to the second one, if they are not in correct order, insert the second element to the correct position of sorted list. If they are in the correct order, insert the element to the end of sorted list. Repeat those steps until all element is in sorted list. I use an integer temp as temporary index to split the list into sorted and unsorted list. After a new unsorted element is inserted to the sorted list, temp += 1. Repeat the steps to the end.

Runtime Performance: As shown in log-log scale, the performance of insertion sort on Almost-sorted permutations is way much better than performance on Uniformly distributed permutations. For almost sorted permutations, we don't need to iterate the sorted list to insert the new unsorted item since most items is "sorted" and we only need to add it to the end of sorted list.

Time Complexity: The best case for insertion sort is an input that is already sorted, in which case it takes linear running time like O(n). The first element in the unsorted list is always inserted in the end of sorted list. The worst case is an input that is in reverse order, in which case every iteration in the inner loop will iterate and shift the entire sorted list before inserting the unsorted element. The worst case running time is O(n^2)

Spin the bottle sort



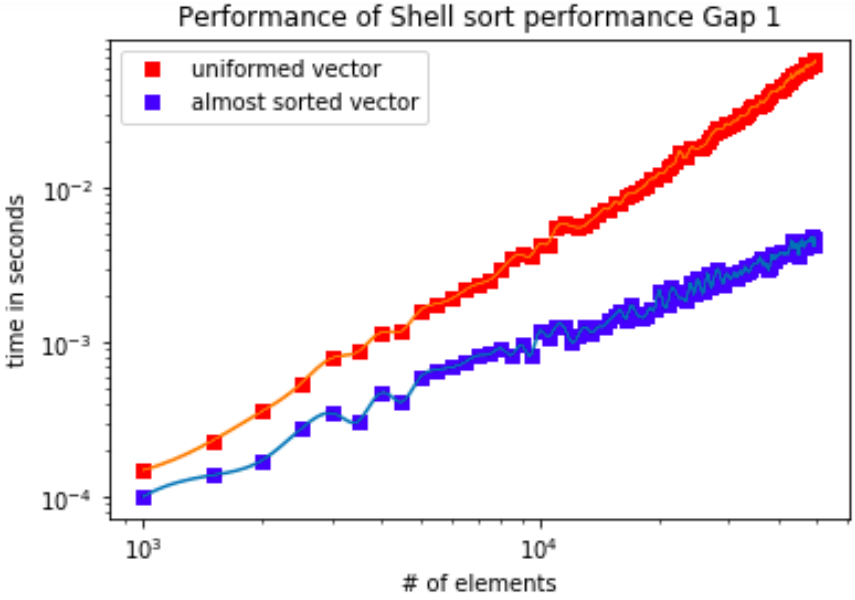
Spin the bottle sort Analysis

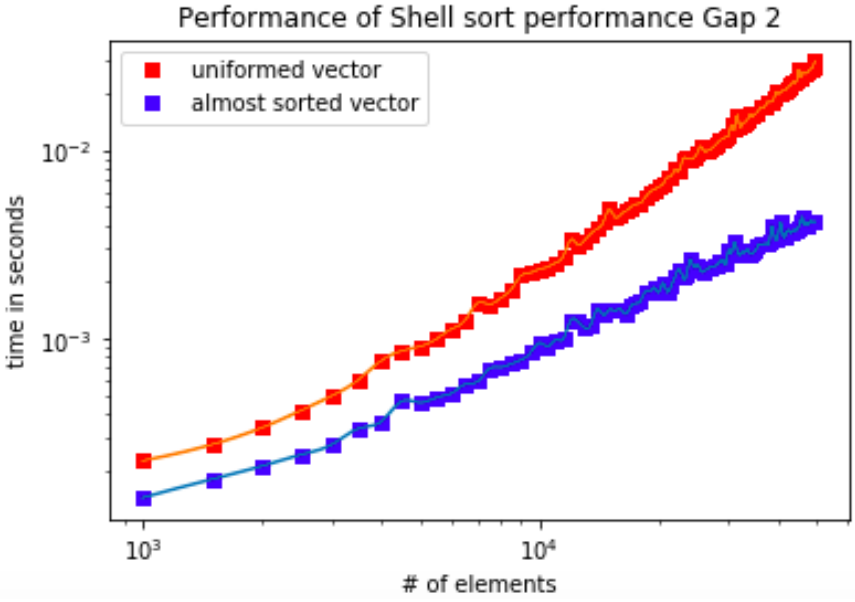
Implementation: Iterate through the unsorted list. Choose an element uniformly and independently at random from the list and compare the element with the current element. If they are in incorrect order, swap them. If not, continue the iteration. Repeat the steps until the list is sorted.

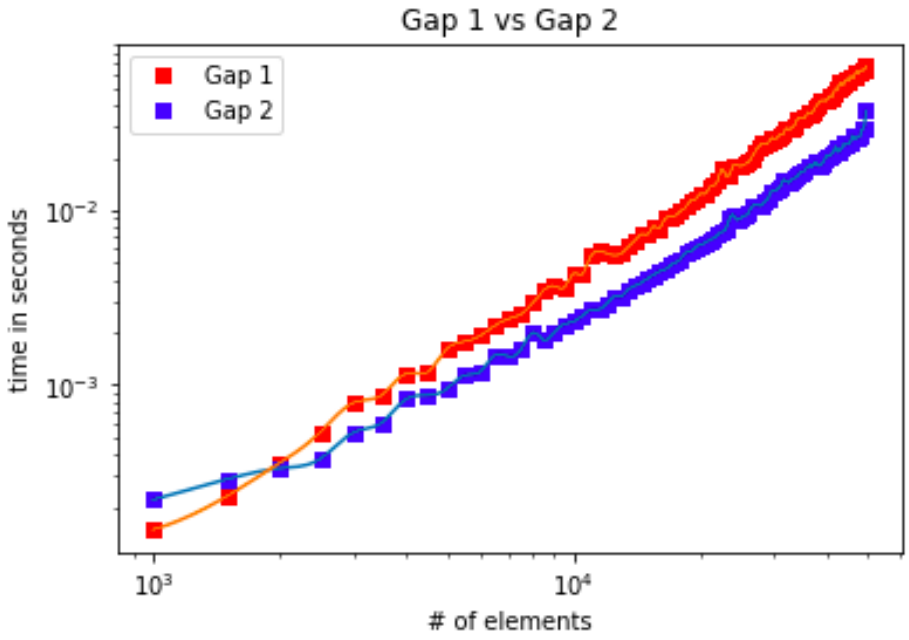
Runtime Performance: As shown in log-log scale, the performances of spin the bottle sort on Uniformly distributed permutations and Almost-sorted permutations are almost the same since the efficiency of this algorithm largely based on randomness. We are comparing each item in unsorted list with another random element in the list. The permutation has slight effect on performance.

Time complexity: According to test, the best case and worse case for spin the bottle is O (n^2 * log n)

Shell sort


Performance of Shell sort performance Gap 1
uniformed vector
almost sorted vector
time in seconds
10^{-2}
10^{-3}
10^{-4}
10^{3}
10^{4}
# of elements


Performance of Shell sort performance Gap 2
uniformed vector
almost sorted vector
time in seconds
10^{-2}
10^{-3}
10^{3}
10^{4}
# of elements



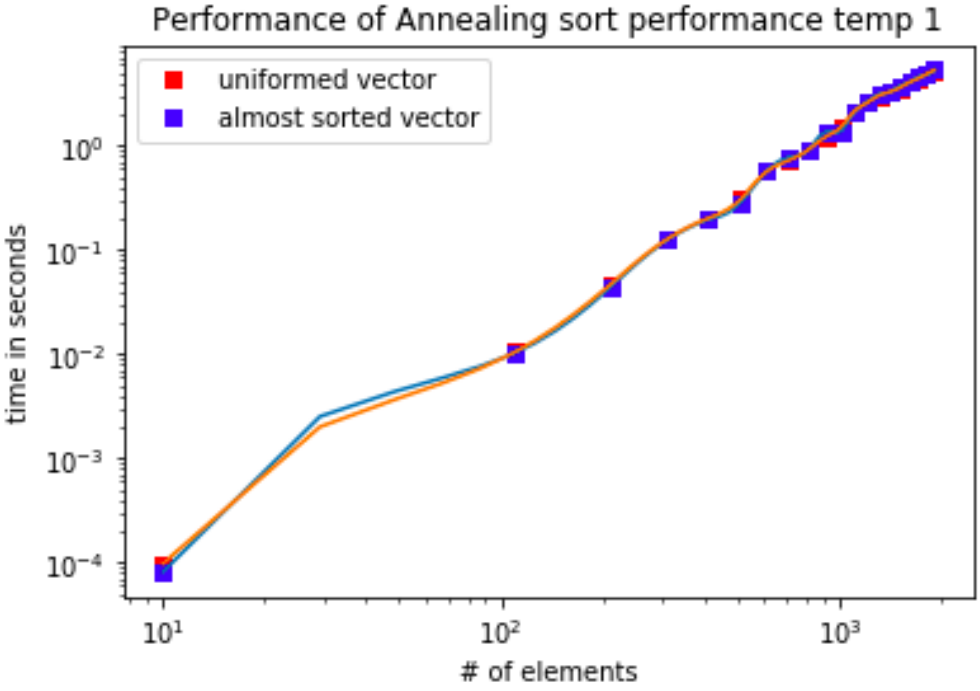
Shell sort Analysis

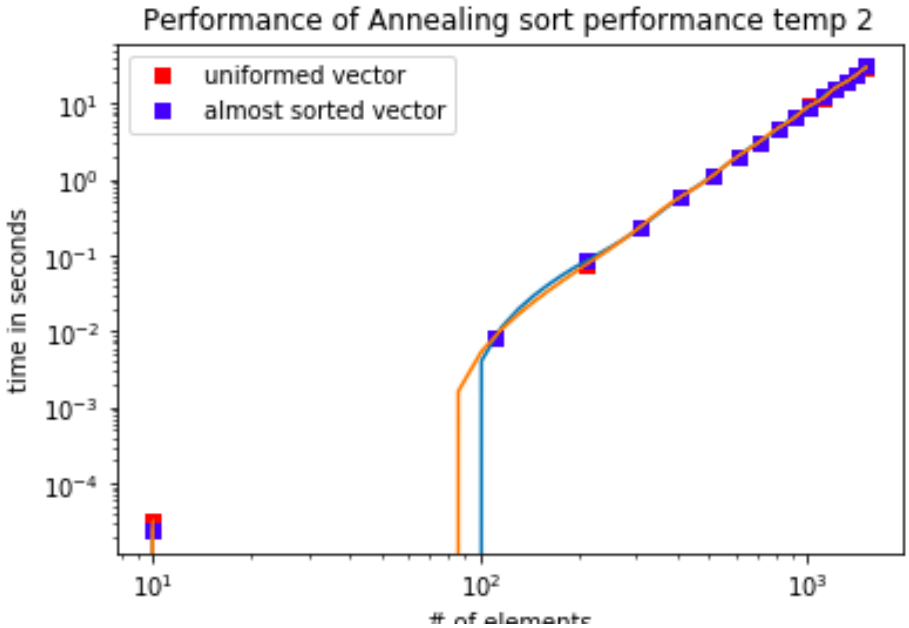
Implementation: Shell sort can be considered as generalization of insertion sort that allows swap of items that are far apart. Given gap sequence, use different gap number to sort the unsorted sequence. Iterate through the gap sequence vector. For each gap, test the element that is "gap" far away. If they are disorder, swap. I test several gaps sets and find two best gap sequence. Gap 1 is (floor(N/2^k), floor(N/2^(k-1))...1) and Gap 2 is ((9*(9/4^(k-1))-4)/5, (9*(9/4^(k-2))-4)/5 ....1)

Runtime Performance: As shown in log-log scale, the performance of insertion sort on Almost-sorted permutations is way much better than performance on Uniformly distributed permutations no matter which what gap we use since as mentioned above, shell sort is generalization of insertion sort. When the list is almost sorted, it takes less time to swap the disorder pair. Also, we can tell from the third plot that Gap2 has better performance than gap 1. We are going to use Gap 2 as best gap for shell sort.

Time complexity: The complexity for Gap 1 is O(n^2). The complexity for gap 2 is also O(n^2)

Annealing sort





Annealing Sort:
Temp1: [N/2-0,N/4 -1….] Reps1: [N/2 - TEMPSIZE]
Temp2: [N/2 -20, N/2 -40…] Reps1:[N – TEMPS -1]

Implementations:
I choose this two sequences: for temp-reps1 1, it will have a smaller value inside the temp but bigger value in reps. For temp-reps 2, it will have a bigger value inside the temp but small value in reps. Both sequence have great successful rate.

Runtime Performance: As shown in log-log scale, the performances of spin the bottle sort on Uniformly distributed permutations and Almost-sorted permutations are almost the same since the efficiency of this algorithm largely based on randomness. We are comparing each item in unsorted list with another random element in the list. They both have great successful rate but temp 2 takes too much time to finish since it has more reps. So I would choose temp-reps 1 pair.

Rank: Annealing-sort > shell sort > insertion sort > spin to bottle > bubble