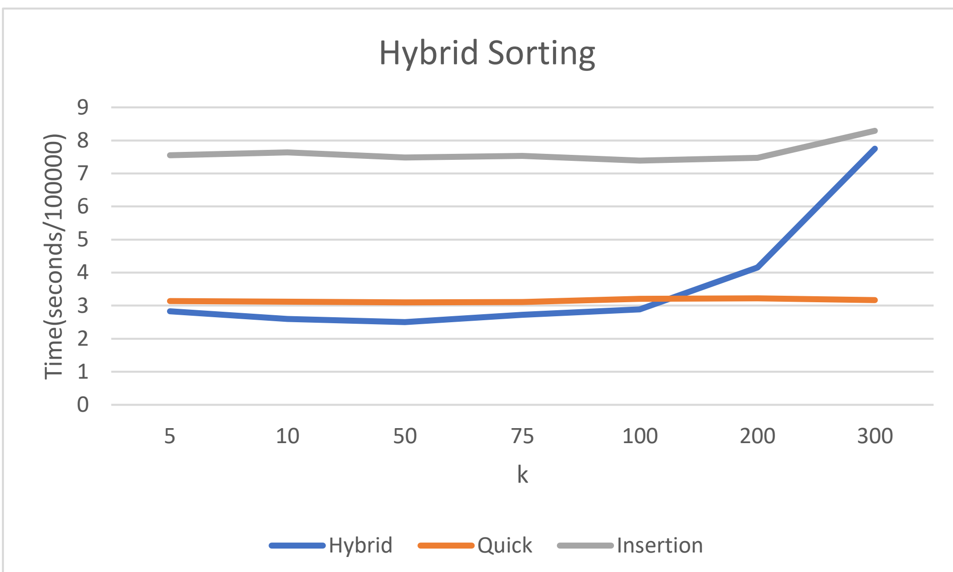
Hypothesis: The optimal array size k to switch from quick sort to insertion sort is k=75.

Methods: I reused my experiment setup from question 1, only adding a hybrid sorting algorithm and test for that algorithm. This time I looped each algorithm 100,000 times, set the sample size (n) to a constant 300 for all tests, and only changed the value k for the hybrid sorting algorithm to switch from quick sort to insertion sort. I used the values: 5, 10, 50, 100, 200 for k. I changed n to 200 and 100 to see if times changed and they did not.

Results:

k	Hybrid Sort	Quick-Sort	Insertion Sort
5	2.8321	3.139	7.5488
10	2.6	3.119	7.635
50	2.503	3.1	7.48
75	2.722	3.11	7.532
100	2.89	3.21	7.39
200	4.151	3.222	7.473
300	7.75	3.164	8.291



Discussion: For the graph, the y-axis is time and the x-axis is k, the value at which the hybrid algorithm switches from quick to insertion sort. The quick sort and insertion sorting algorithms times stay constant as k increases, which is expected as the big(O) for both depend on n, the input size, which stays at a constant 300 for these tests. It is interesting to note that as k increases, hybrid sort starts behaving more like insertion sort. This makes sense because when k = n, hybrid sort only calls insertion sort and does not recurse or behave as quick sort at all. The important point on the graph is where the lowest y value occurs for hybrid sort, which we can see is at about 50. K is around the same as the crossover point for part 1. This is because insertion sort runs faster for array sizes less than 50 and quick sort runs faster for array sizes of greater than 50. Changing n to 200 and 100 did not change the times.

<u>Conclusion:</u> Under the conditions tested, the optimal array size k to switch from using quick sort to insertion sort is k=50.

```
#include <time.h>
#include <ctime>
#include <cstdlib>
#include <iostream>
#include <string>
#include <vector>
#include <bits/stdc++.h>

using namespace std;

/*
 *
 */

/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int low, int high)
{
    int i, key, j;
    for (i = low + 1; i < high; i++)
    {
        key = arr[i];
        j = i - 1;

        /* Move elements of arr[0..i-1], that are
        greater than key, to one position ahead
        of their current position */
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

//QUICK SORT FUNCTIONS START HERE
// A utility function to swap two elements
void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

/* This function takes last element as pivot, places
the pivot element at its correct position in sorted
array, and places all smaller (smaller than pivot)
```

```c
/* This function takes last element as pivot, places
the pivot element at its correct position in sorted
array, and places all smaller (smaller than pivot)
to left of pivot and all greater elements to right
of pivot */
int partition (int arr[], int low, int high)
{
    int pivot = arr[high]; // pivot
    int i = (low - 1); // Index of smaller element

    for (int j = low; j <= high - 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++; // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

/* The main function that implements QuickSort
arr[] --> Array to be sorted,
low --> Starting index,
high --> Ending index */
void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
        at right place */
        int pi = partition(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

/* The main function that implements QuickSort
arr[] --> Array to be sorted,
low --> Starting index,
high --> Ending index */
void hybridSort(int arr[], int low, int high)
{
```

```
high --> Ending index */
void hybridSort(int arr[], int low, int high)
{
    //CHANGE THIS FOR EXPERIMENTS
    if(high - low < 300){
        insertionSort(arr,low,high);
        return;
    }
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
        at right place */
        int pi = partition(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        hybridSort(arr, low, pi - 1);
        hybridSort(arr, pi + 1, high);
    }
}



int main(int argc, char** argv) {

    //Seed for random number
    srand(time(NULL));

    constexpr int SAMPLE_SIZE = 300;
    constexpr int NUM_TESTS = 10000;
    constexpr int RANGE = SAMPLE_SIZE*10;

    int array[SAMPLE_SIZE];


    //Time testing for data allocation itself
    clock_t ds_time = clock();
    for(int i = 0; i < NUM_TESTS; i++){
        for (int i=0;i<SAMPLE_SIZE;i++){
            array[i] = rand() % RANGE;
        }
    }
    clock_t df_time = clock() - ds_time;

    cout << ((double) df_time) / (double) CLOCKS_PER_SEC << endl;

    //Actual time testing for quicksort
```

```
int main(int argc, char** argv) {

    //Seed for random number
    srand(time(NULL));

    constexpr int SAMPLE_SIZE = 300;
    constexpr int NUM_TESTS = 10000;
    constexpr int RANGE = SAMPLE_SIZE*10;

    int array[SAMPLE_SIZE];


    //Time testing for data allocation itself
    clock_t ds_time = clock();
    for(int i = 0; i < NUM_TESTS; i++){
        for (int i=0;i<SAMPLE_SIZE;i++){
            array[i] = rand() % RANGE;
        }
    }
    clock_t df_time = clock() - ds_time;

    cout << ((double) df_time) / (double) CLOCKS_PER_SEC << endl;

    //Actual time testing for quicksort
    clock_t s_time = clock();
    for(int i = 0; i < NUM_TESTS; i++){
        for (int i=0;i<SAMPLE_SIZE;i++){
            array[i] = rand() % RANGE;
        }
        quickSort(array,0,SAMPLE_SIZE);
    }
    clock_t f_time = clock() - s_time - df_time;
    cout << "Quick Sort Time: "
         << ((double) f_time) / (double) CLOCKS_PER_SEC
         << " seconds" << endl;

    //Time testing for insertion sort
    s_time = clock();
    for(int i = 0; i < NUM_TESTS; i++){
        for (int i=0;i<SAMPLE_SIZE;i++){
            array[i] = rand() % RANGE;
        }
        insertionSort(array,0, SAMPLE_SIZE);
    }
    f_time = clock() - s_time - df_time;
    cout << "Insertion Sort Time: "
         << ((double) f_time) / (double) CLOCKS_PER_SEC
```

```cpp
    cout << ((double) df_time) / (double) CLOCKS_PER_SEC << endl;

    //Actual time testing for quicksort
    clock_t s_time = clock();
    for(int i = 0; i < NUM_TESTS; i++){
        for (int i=0;i<SAMPLE_SIZE;i++){
            array[i] = rand() % RANGE;
        }
        quickSort(array,0,SAMPLE_SIZE);
    }
    clock_t f_time = clock() - s_time - df_time;
    cout << "Quick Sort Time: "
         << ((double) f_time) / (double) CLOCKS_PER_SEC
         << " seconds" << endl;

    //Time testing for insertion sort
    s_time = clock();
    for(int i = 0; i < NUM_TESTS; i++){
        for (int i=0;i<SAMPLE_SIZE;i++){
            array[i] = rand() % RANGE;
        }
        insertionSort(array,0, SAMPLE_SIZE);
    }
    f_time = clock() - s_time - df_time;
    cout << "Insertion Sort Time: "
         << ((double) f_time) / (double) CLOCKS_PER_SEC
         << " seconds" << endl;

    //Time testing for hybrid sort
    s_time = clock();
    for(int i = 0; i < NUM_TESTS; i++){
        for (int i=0;i<SAMPLE_SIZE;i++){
            array[i] = rand() % RANGE;
        }
        hybridSort(array,0, SAMPLE_SIZE);
    }
    f_time = clock() - s_time - df_time;
    cout << "Hybrid Sort Time: "
         << ((double) f_time) / (double) CLOCKS_PER_SEC
         << " seconds" << endl;

}
```