Benjamin Miller

Hybrid Sorting

Hypothesis:

I believe that the most efficient k will be n = 65, as found in question 1. If it is not, I expect that it will be below 65, rather than higher. I expect to find that the size of the array does not impact k.

Methods:

For this experiment I used CLion by Jetbrains to run the code. I created a new project in c++ 14 and put my code into the file that ended with ".cpp". I did not do anything special for the compiler or any optimizations. Once the code was added, I simply hit the run button to get the outputs. For the code in the main function I did these steps:

1. Created a loop to run 1,000 times and started the timer, so each test would sort 1,000 arrays
2. Created an array of size 100 in each iteration, and filled it with random numbers from 1-100
3. Called the hybrid sort function, passing k = 1 to k = 65, manually changing the k after each run, and recorded the time output
4. Repeat step 3, with array of size 25

The auto generated makefile:

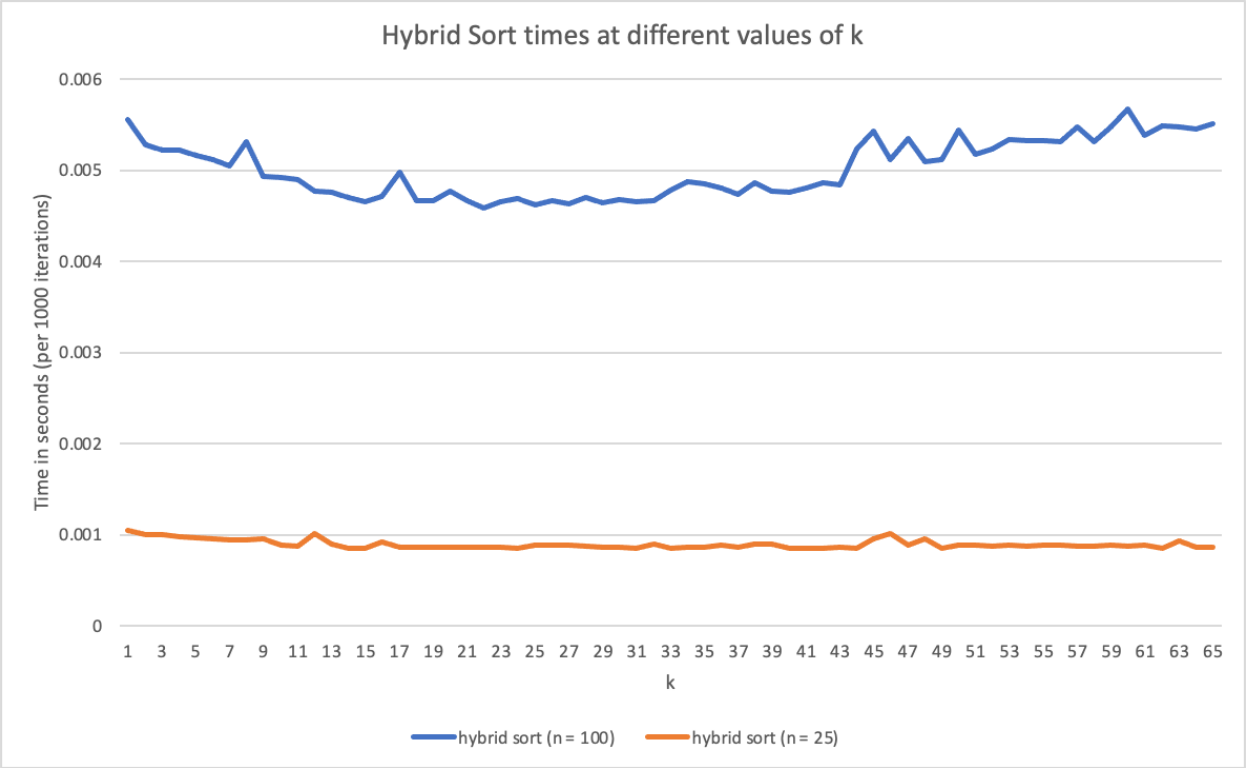
```
cmake_minimum_required(VERSION 3.17)
project(431Question2)
set(CMAKE_CXX_STANDARD 14)
add_executable(431Question2 main.cpp)
```

```cpp
#include <iostream>
#include <ctime>
#include <vector>

using namespace std;


// ################## Hybrid Sort ##############################
// https://www.geeksforgeeks.org/advanced-quick-sort-hybrid-algorithm/

void insertion_sort(int arr[], int low, int n)
{

    for(int i=low+1;i<n+1;i++)
    {
        int val = arr[i] ;
        int j = i ;
        while (j>low && arr[j-1]>val)
        {
            arr[j]= arr[j-1] ;
            j-= 1;
        }
        arr[j]= val ;
    }

}

//The following two functions are used
// to perform quicksort on the array.


// Partition function for quicksort
```

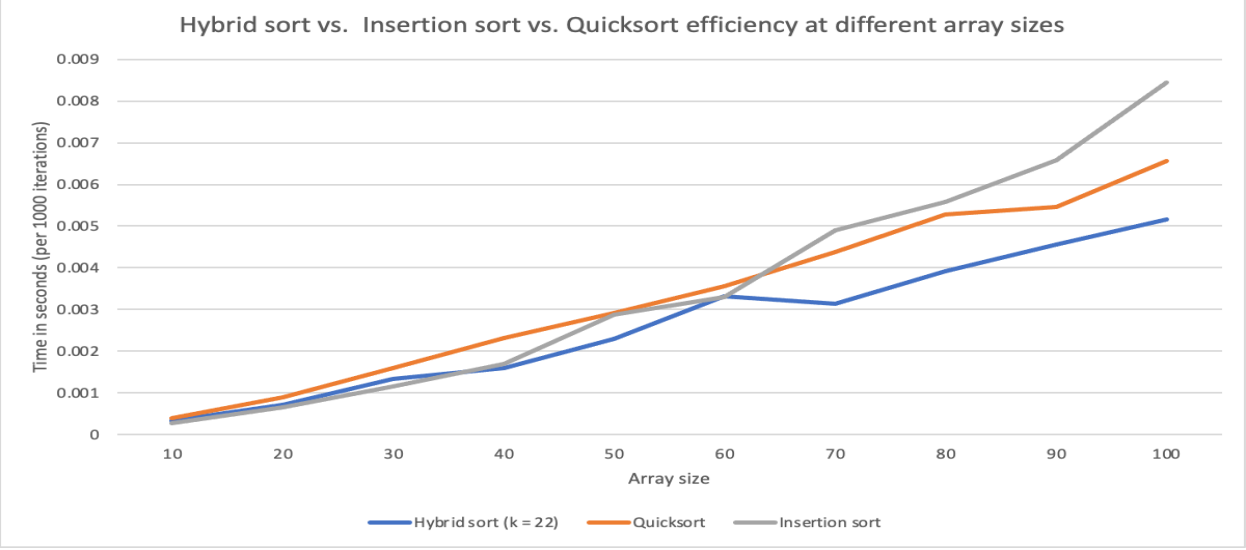
```
int partition(int arr[], int low, int high)
{
    int pivot = arr[high] ;
    int i ,j;
    i = low;
    j = low;

    for (int i = low; i < high; i++)
    {
        if(arr[i]<pivot)
        {
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
            j += 1;
        }
    }

    int temp = arr[j];
    arr[j] = arr[high];
    arr[high] = temp;

    return j;
}


// Function to call the partition function
// and perform quick sort on the array


void quick_sort(int arr[], int low,int high)
{
    if (low < high)
    {
        int pivot = partition(arr, low, high);
        quick_sort(arr, low,  high: pivot-1) ;
        quick_sort(arr,  low: pivot + 1, high) ;


    }
}

// Hybrid function -> Quick + Insertion sort

void hybrid_quick_sort(int arr[], int low, int high, int k)
{
    while (low < high)
    {

        // If the size of the array is less
        // than threshold apply insertion sort
        // and stop recursion

        if (high-low + 1 <= k)
        {
            insertion_sort(arr, low, high);
            break;
        }

        else

        {
            int pivot = partition(arr, low, high) ;
```

```
            // Optimised quicksort which works on
            // the smaller arrays first

            // If the left side of the pivot
            // is less than right, sort left part
            // and move to the right part of the array

            if (pivot-low<high-pivot)
            {
                hybrid_quick_sort(arr, low,  high: pivot - 1, k);
                low = pivot + 1;
            }
            else
            {

                // If the right side of pivot is less
                // than left, sort right side and
                // move to the left side

                hybrid_quick_sort(arr,  low: pivot + 1, high, k);
                high = pivot-1;
            }

        }

    }
}
// ############################################################

 // https://stackoverflow.com/questions/26051101/how-
 // to-fill-an-array-with-random-values-from-a-range-duplicates-are-ok/26051286
int main() {
    int loops = 1000;

    for (int k = 1; k < 2; k++) {
        std::clock_t start_time = std::clock();
        srand(time(0));

        for (int i = 0; i < loops; i++) {
            int size = 100;

            int data[size];
            std::generate(data,  last: data + sizeof(data) / sizeof(int),  gen: [&]() {
                return rand() % 100;
            });

            int n = sizeof(data) / sizeof(data[0]);
            int k = 22;

            hybrid_quick_sort(data,  low: 0,  high: n - 1, k);

        }

        std::clock_t tot_time = std::clock() - start_time;
        std::cout << "Time at " << k << ": "
                  << ((double) tot_time) / (double) CLOCKS_PER_SEC
                  << " seconds" << std::endl;

    }
    return 0;
}
```

Results:



This graph above shows the sort times for a hybrid sorting algorithm when the size of the array is equal to 100 (blue) and 25 (orange). The lowest point for n = 100 is at k = 22. The lowest points for n = 25 are at 20 < k < 30. The time is in seconds and represents 1000 total iterations completed, where for each iteration an array is sorted.



The three sorting algorithms compared. The graph shows the time in milliseconds that it takes for each algorithm to sort an array of n size 1000 total times.

Discussion:

I was very surprised once again by the results that I found. The best k value was roughly at 22, which is nowhere near the 65 that I expected. These results are more what I had expected from problem 1. I did not face any real challenges for the data collection on this question. I am not sure why the crossover points would not be the same. My best guess is that since you have to now make multiple function calls no matter which algorithm you use, that quicksort now is better at a lower value. The comparison between the three sort algorithms was what I expected however, and represented what should happen. Lastly, the size of the array seemed to have little impact aside from causing time to have a slightly wider range.

Conclusions:

Under the conditions tested, the most efficient k for a hybrid sort was at k = 22. The size of the array did not seem to impact the k value significantly. Lastly, hybrid sort produces the fastest runtimes for array sizes larger than 60, and insertion sort produces the slowest when size is larger than 60, and quicksort is between the two. For array sizes less than 60, quicksort is the slowest, while hybrid and insertion trade off being the fastest.