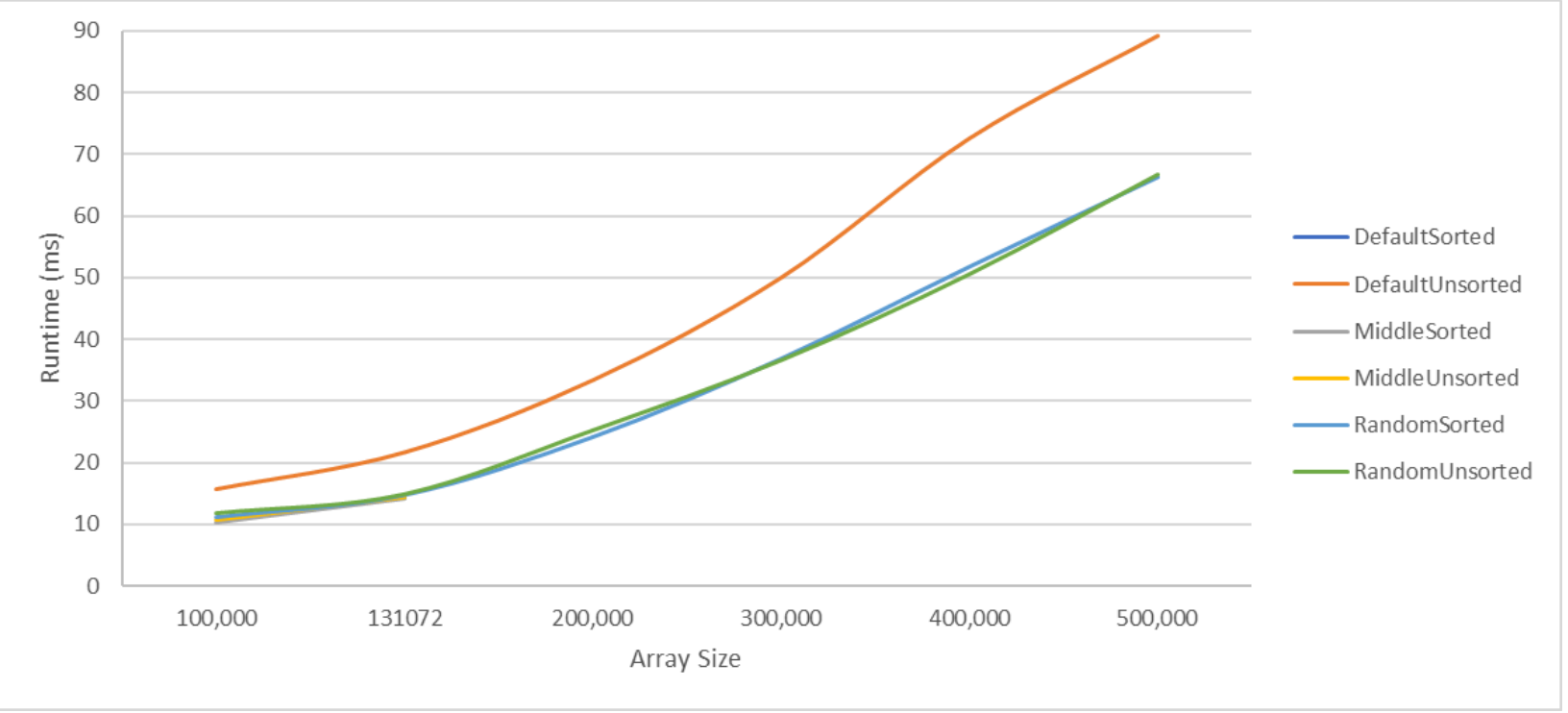
Project 2
Zayne Bonner

Project 2 Average Runtimes | time in **ms**

Size	Default Sorted	Default Unsorted	Middle Pivot Sorted	Middle Pivot Unsorted	Random Sorted	Random unsorted
100,000	32,725	15.63	10.2	10.72	11.1	11.7
200,000	124,709	33.3	took too long	took too long	24.1	25.2
300,000	269,480	49.92	took too long	took too long	36.8	36.6
400,000	Took too long	72.5	took too long	took too long	51.7	50.6
500,000	Took too long	89.1	took too long	took too long	66.2	66.8
131072	5566.3	21.6	14.1	14.3	14.7	14.8



For this Project, as I will mention soon, quicksorting within the middle half is not fully functional as data gets big, hence the small, cut off lines and the inclusion of the extra dataset that may skew the line graph a little.

For this experiment, we are quick sorting arrays that are unsorted and arrays that are sorted. I used a standard quicksort with the last number being the pivot, a quicksort with a randomized pivot, and a quicksort with the pivot within the middle half of the array indexes. I programmed these methods using C++ within VSC. The programming language used I predicted would run much faster than python, but may have contributed to the one fatal error that seems unavoidable. This will be discussed later. From our knowledge of quicksort, we should already be able to predict the outcomes of each. Standard quicksort has a time complexity of O(n log n), and a worst case of O(n^2). We should be able to predict that an unsorted array should fall closer to O(n log n), and the sorted array, being the worst case, should come out to have a time complexity of O(n^2). A random quicksort helps avoid the worst case, which is essentially only encountered with a sorted array. We should see a consistent O(n log n) throughout our

experiment, even with a sorted array, since the pivot is grabbed at random, and is much less likely to be the largest value every time. In a middle-focused pivot, with an unsorted array we should expect close to the results of the random pivot quicksort, which is O(n log n). We should be able to predict that with a sorted array, since the middle half is closer to the median on average, we should be able to get very close to an O(n log n) and slightly shorter time complexity than even the random pivot. Our experiment will confirm or deny these hypotheses. For coding these methods, I based my code off of the GeeksForGeeks quicksort, and modified it to match each criteria. For the testing, I individually ran each type of quicksort with a for-loop. Within the for-loop, I would generate a random unsorted array, then I would run the quicksort of the array, then run it again with the same array that has then been sorted by the previous run. Each iteration would have a new array that is quicksorted unsorted, then sorted. This method is efficient, until we get to the middle pivot. As our data got bigger, I found that the very-efficient middle-based pivot did not work for any array bigger than 131072 numbers. I could not deduce the exact issue, I asked around, and even found a few others had the same issue. I increased the stack size heavily and maintained and optimized the code the best I could while keeping the format the same. I could not deduce any exact issue, whether it be poor code, memory utilization, or anything else. This quicksort runs indefinitely with 0 output. This unfortunately really hurts our data. Our goal is to see which is the fastest.

After running the experiment, we got a lot of data as seen above. For standard quicksort, we can see that it is slow, but you may notice that the standard quicksort of the sorted array is not visible on the graph. This is because, as predicted, it is much slower due to it being the exact worst case. We can see that both middle-based pivot quicksorts are small lines due to the errors mentioned in the introduction. We can see however, that random quicksort is comparable at smaller data, then as it has more data, continues much more efficiently than the standard quicksort. The random quicksort always ran faster than the standard quicksort, and both random quicksorts ran almost equivalently. I would predict that if the middle-based pivot would run on bigger data, it would consistently run just below or equal to the random quicksorts on average for unsorted arrays, and run noticeably faster on sorted arrays as data got bigger, since it would never run the worst case ever, but would still have some bad cases in choosing the pivot. The most effective for a sorted array would be to use n/2 as the pivot, since that would effectively be the median and provide the most optimal split, but that is not what this is about. I considered adding an exception that if the partition function can not find a good pivot within 100 attempts, just default to the middle (n/2) element, however I feel that would produce incredibly skewed answers and that is not what was asked of the experiment.

Lastly, given the closed set of data, the comparisons are near, but not quite what I would expect. In a completely randomized, unsorted array, it would be expected that each type of quicksort would be comparable. I would predict that any element that the random quicksort would grab as the pivot is equally likely to be the largest as the standard quicksort grabbing the last index. They would, in theory, consistently be equally likely to be the largest, and thus comparable, but as the data grew, the random pivot was always faster than the default quicksort.

The default quicksort produced accurate runtimes. With the array sorted, the worst case was executed, and the runtime was very,very big and grew rapidly (O(n^2)). With an unsorted array, the largest element was not consistently chosen as the pivot, and thus was much closer to an O(n log n) runtime. For a random quicksort, both performed equivalently, which was expected. The pivot being chosen randomly allowed for the largest, smallest, or anything between the two to be chosen equally, no matter how sorted the array is. With the experiment we can see that the runtime was in line with O(n log n). For the middle-based index, it performed on par with the random quicksort for an unsorted array. It was slightly faster than the random quicksort when the array was sorted. This is due to the middle-based index quicksort excluding the worst cases from the possible values when selecting the pivot. This was of course when the data was small, and it was executable. If this version of the code worked for larger data, we would see that with a sorted array, it would likely show to be consistently faster than the random quicksort when sorting an already sorted array, and be comparable in a random array. Since the middle-based quicksort does not work as data gets bigger, and it always does, the random quicksort would be recommended in any case. If the other quicksort functioned as it would suggest, it would be recommended to utilize it instead, since during the case of a sorted array the worst case is completely eliminated.

Appendix:

Within tasks.json, it is dire that the stack size is increased to suit the large arrays used within the standard quicksort.

Here is the full C++ program; written within VSC. Raw file is also attached in submission.

```
#include <bits/stdc++.h>
#include <chrono>
using namespace std;
//Zayne Bonner 800756759


int partitionRandom(vector<int>& arr, int low, int last){
    int randomIndex = low +  rand() % (last - low + 1);//grab a random
pivot < size of the last value of array to be sorted
    swap(arr[randomIndex], arr[last]);
    int pivot = arr[last]; //last item in array = pivot

    int i = low-1;//smaller element index

    for (int j = low; j <= last - 1; j++) {
        if (arr[j] < pivot) {
            i++;
```

```
            swap(arr[i], arr[j]); //less than pivot => move the indexed
element to left
        }
    }

    swap(arr[i + 1], arr[last]); //move the pivot to be behind the smaller
elements
    return i + 1;
}
int partitionMiddle(vector<int>& arr, int low, int last){
    int randomIndex;
    int midLow = low + (last - low)/4;
    int midHigh = low + 3*(last-low)/4;
    do{
        randomIndex = low + rand() % (last - low +1);
    }while(randomIndex < midLow || randomIndex > midHigh);

    swap(arr[randomIndex], arr[last]);
    int pivot = arr[last]; //last item in array = pivot
    int i = low-1;//smaller element index
    for (int j = low; j <= last - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]); //less than pivot => move the indexed
element to left
        }
    }

    swap(arr[i + 1], arr[last]); //move the pivot to be behind the smaller
elements
    return i + 1;


}


int partition(vector<int>& arr, int low, int last) {

    int pivot = arr[last]; //last item in array = pivot
```

```
    int i = low-1;//smaller element index

    for (int j = low; j <= last - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]); //less than pivot => move the indexed
element to left
        }
    }

    swap(arr[i + 1], arr[last]); //move the pivot to be behind the smaller
elements
    return i + 1;
}


void quickSortRandom(vector<int>& arr, int low, int last){
    if (low < last) {

        int pi = partitionRandom(arr, low, last);

        quickSortRandom(arr, low, pi - 1); //recursively sort left of
pivot
        quickSortRandom(arr, pi + 1, last); //recursively sort right of
pivot
    }
}
void quickSortMiddle(vector<int>& arr, int low, int last){
    if (low < last) {

        int pi = partitionMiddle(arr, low, last);

        quickSortMiddle(arr, low, pi - 1); //recursively sort left of
pivot
        quickSortMiddle(arr, pi + 1, last); //recursively sort right of
pivot
    }
}

void quickSort(vector<int>& arr, int low, int last/*,int test*/) {
```

```
        /*for (int i = 0; i < test; i++) {
        cout << arr[i] << " ";
        }
        cout << endl;*/
    if (low < last) {

        int pi = partition(arr, low, last);//pivot index

        quickSort(arr, low, pi - 1); //recursively sort left of pivot
        quickSort(arr, pi + 1, last); //recursively sort right of pivot
    }
}

int main() {
    const int size = 131073;//131073
    const int min = 1;
    const int max = 1000000;
    srand(time(0));
    random_device rd;
    mt19937 eng(rd());
    uniform_int_distribution<> distr(min, max);

    vector<int> arr(size);

    for (int i = 0; i < size; i++)
        arr[i] = distr(eng);

    int n = arr.size();
    vector<int> arr2=arr;
    vector<int> arr3=arr;
    int z;
for(z=0;z<11;z++){//calls whatever is within 11 times, since the first
execution is always an outlier that involves extra initializations.
    uniform_int_distribution<> distr(min, max);

    vector<int> arr(size);

        for (int i = 0; i < size; i++)
            arr[i] = distr(eng);
```

```
}//put this for-loop around any quicksort call function and timer that you
wish to call 10x times
//      ################################################
//         Array Quicksort Calls:

//Standard Quicksort
    auto start = chrono::high_resolution_clock::now();
        quickSort(arr, 0, n-1);

    auto stop = chrono::high_resolution_clock::now();
    chrono::duration<double,milli> elapsed = stop-start;
        cout << elapsed.count()<<" milliseconds - Unsorted
Quicksort"<<endl;

    start = chrono::high_resolution_clock::now();
       //quickSort(arr, 0, n-1);

    stop = chrono::high_resolution_clock::now();
    elapsed = stop-start;
        cout << elapsed.count()<<" milliseconds - sorted Quicksort"<<endl;
//Middle Quicksort
    start = chrono::high_resolution_clock::now();
        quickSortMiddle(arr2, 0, n-1);

    stop = chrono::high_resolution_clock::now();
    elapsed = stop-start;
        cout << elapsed.count()<<" milliseconds - Unsorted Quicksort with
mid-range pivot"<<endl;


    start = chrono::high_resolution_clock::now();
        quickSortMiddle(arr2, 0, n-1);

    stop = chrono::high_resolution_clock::now();
    elapsed = stop-start;
       cout << elapsed.count()<<" milliseconds - sorted Quicksort with
mid-range pivot"<<endl;

//Random Quicksort
    start = chrono::high_resolution_clock::now();
```

```
        quickSortRandom(arr3, 0, n-1);

    stop = chrono::high_resolution_clock::now();
        elapsed = stop-start;

        cout << elapsed.count()<<" milliseconds - Unsorted Quicksort with
random  pivot"<<endl;
    start = chrono::high_resolution_clock::now();
        quickSortRandom(arr3, 0, n-1);

    stop = chrono::high_resolution_clock::now();
    elapsed = stop-start;
        cout << elapsed.count()<<" milliseconds - sorted Quicksort with
random  pivot"<<endl;



/*
    for(z=0;z<10;z++){
    uniform_int_distribution<> distr(min, max);

    vector<int> arr(size);

        for (int i = 0; i < size; i++)
            arr[i] = distr(eng);

    }
    cout << "Sorted Array" << endl;
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
    for (int i = 0; i < n; i++) {
        cout << arr2[i] << " ";
    }
    cout << endl;
    for (int i = 0; i < n; i++) {
        cout << arr3[i] << " ";
    }
    cout << endl;
```

```
*/
    return 0;
}
```