Enhancing Quicksort

Dr. John Matta
Ryan Bender
CS 340 – 002

Introduction:

The purpose of this project is to compare the speeds of modified quicksort algorithms with different pivoted elements and understand how to manage large amounts of data. Quicksort is an extremely time efficient sorting algorithm that uses the divide and conquer technique to provide a rapidly sorted array. For this project, I tested three different variations of the quicksort algorithm. The first variation used the last element as a pivot, the second used the middle, and the third used a random element. Besides changing the pivot, these three algorithms were near identical.

My approach to this project was to first get one quicksort algorithm, using the last element as the pivot, working to make sure the program could run through 500,000 elements. Once the basic algorithm was working, I was able to create modified versions of the original partition function to allow for the pivot to be the middle and a random element. Once the pivot element is selected, it is swapped to the end of the array. This method keeps the properties of selecting the middle or random element, but it allows the same logic to be used for all three approaches. The three quicksort algorithms are the same setup, but each calls a different partition function, making for an easier following of the code.

The theoretical time complexities of each algorithm are the same due to the logic of each following the same process. This means that the average time complexity for each algorithm is $nlog(n)$ and the worst-case complexity is n^2. Despite each time complexity being the same, the runtimes can vary drastically depending on the pivot's value. These runtimes will be examined more during the results section of this report.

For this project, I used C++ as my programming language due to the nature of the project's large amount of data. C++ allows for extremely effective use of memory, as it provides the programmer with the ability to allocate and deallocate memory. This is a massive benefit for C++ when the data is large. Other languages, like Python, perform memory management automatically, leaving for less control for programmer. Additionally, C++ is a compiled language meaning the code is compiled down to machine code. This allows the computer to run the program faster, saving time when dealing with big data. Since I used C++, the program ran much faster than it would of using other languages, as well as put the job of managing memory into my control.

Sorting large amounts of data can be a complicated job when it comes to having enough allocated memory for all the necessary tasks. In this project, one of the most common issues that I ran into was not having enough allocated memory. I received countless segmentation faults which complicated the programming process. My first issue was allocating the initial memory for declaring the arrays, which was an easy fix with the use of pointers. Using pointers allowed the array to be spread out in memory rather than being sequential. The next issue I ran into was sorting the pre-sorted arrays while using the last index as the pivot. When the array is already sorted, the partitions will not need to be sorted since everything is in order already. This causes an issue when the pivot is in the beginning or end of the array because the partitions are completely off balance. With the partitions being too off balance, the program used too much memory causing more segmentation faults. When the pivot is the end, the entire array is the left partition, and the right side is empty. Since the entire remaining array is the left partition, your number of partitions will be n, thus the time complexity is n^2. This issue only happens when the pivot is on one of the ends of the array.

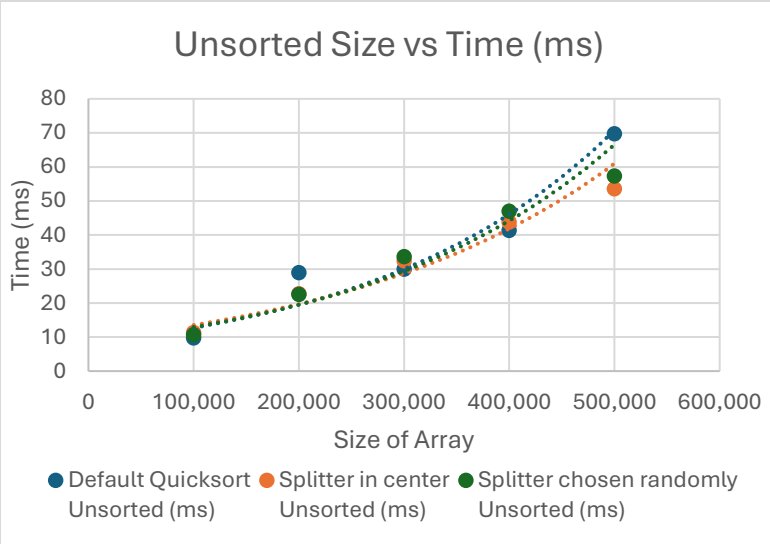
Results:

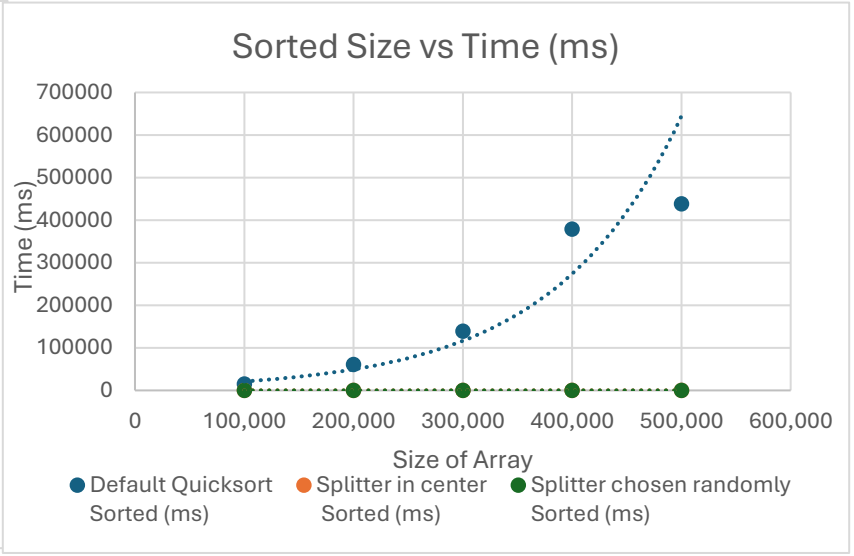
This project was designed to show how changing the pivot of a quicksort algorithm would change the runtime of a program. To display this, I sorted arrays of different sizes ranging from 100,000 to 500,000. These sizes were picked to show the difference when a large amount of data was being sorted. First, I ran through each of the three different variations, first without being sorted and then when it was already sorted. The data below shows the times for each algorithm.

Size	Default Quicksort Sorted/Unsorted (ms)	Splitter in center half Sorted/Unsorted (ms)	Splitter chosen randomly Sorted/Unsorted (ms)
100,000	15156.5 / 9.7554	4.066 / 11.3332	6.0319 / 10.7012
200,000	61074.2 / 28.9225	9.3267 / 22.6849	15.1546 / 22.4876
300,000	138998 / 29.9649	12.6179 / 32.3585	20.5489 / 33.6547
400,000	378935 / 41.3169	17.0506 / 43.7926	29.4494 / 46.9815
500,000	438041 / 69.6931	20.7198 / 53.5731	34.6123 / 57.307

For the unsorted arrays, all the variations of quicksort were relatively close together, but as the size grew, the algorithm that used the end as the pivot started to fall behind. This was a consistent trend despite how many times I would run the program. The center and random pivot ran nearly identical results, only being off by a couple of milliseconds with each size. On the contrary, the pre-sorted arrays were not sorted with the same speed for

each variation. The middle and random pivots were quicker when the array was already sorted, unlike the end pivot. The disparities between variations were much higher than the unsorted.





The graphs above display the relationship between the size of the arrays and how long each algorithm took to fully sort each dataset. As stated early, the unsorted algorithms ran close together, but the sorted algorithm took much longer for the end of array pivot. For the middle and random pivots, the sorted algorithm ran in a similar time to the unsorted. This is because the partition of the middle pivot was equally balance with half of the array being larger than the pivot and half being smaller. When the middle pivot ran through the sorted array, the partitions were always equal length, causing it to hit the best-case runtime. The random pivot performed well because it chose a random element, it had a high chance of being in the middle of the dataset.

Comparing Runtimes:

The runtimes were close to what I was expecting, and they were close to the average runtime the unsorted graph showed a slow exponential growth like that of $nlog(n)$. Due to the nature of how quicksort is set up, sorting an unsorted array gives you an equal chance of getting a number with its value near the median of the array no matter what the value of the pivot is. The data follows this idea because each algorithm was competing in a close race for being the fastest. Unfortunately, when working with large amounts of data, the data may be partially sorted already. Because of this, you must design an algorithm that runs quickly not matter what the data is. The largest limitation to the speed of quicksort is how many partitions it must make. Ideally, it will make log(n) partitions, and the program would run at the fastest speed. When the algorithm is already sorted and its pivot is the first or

last element, the array will make n partitions causing its runtime to be n^2. This is a massive delay in the programs speed compared to $nlog(n)$. On the other two variations of quicksort, the element is either in the middle already and you have log(n) partitions, or it's a random element that will have between log(n) and n^2 partitions. This is the reason why pivoting in the middle is the most effective method of Quicksorting.

Code Used in Project:

EnhancingQuickSort.cpp

```
#include <iostream>
#include <chrono>
#include "EnhancingQuickSort.h"
using namespace std;

int main(){
  int* unsorted1 = new int[100000];
  int* unsorted2 = new int[200000];
  int* unsorted3 = new int[300000];
  int* unsorted4 = new int[400000];
  int* unsorted5 = new int[500000];

  int* sorted1 = new int[100000];
  int* sorted2 = new int[200000];
  int* sorted3 = new int[300000];
  int* sorted4 = new int[400000];
  int* sorted5 = new int[500000];
  populateSorted(sorted1, sorted2, sorted3, sorted4, sorted5);

  /*

    End Index

  */

  // This rounds of sorting uses the last index as a pivot
  populateRandoms(unsorted1, unsorted2, unsorted3, unsorted4, unsorted5);
```

```
cout << "\n\n\n\n Back Pivot"<< endl;
cout <<  endl << "Unsorted Array 1:" << endl;
auto start = chrono :: high_resolution_clock :: now();
quickSort(unsorted1, 0, 100000 - 1);
auto stop = chrono :: high_resolution_clock :: now();
chrono::duration<double> timeTaken = stop - start;
sampleArray(unsorted1);
cout << "Time taken: " << timeTaken.count() * 1000  << " milliseconds" << endl;

cout << endl << "Unsorted Array 2:" << endl;
start = chrono :: high_resolution_clock :: now();
quickSort(unsorted2, 0, 200000 - 1);
stop = chrono :: high_resolution_clock :: now();
timeTaken = stop - start;
sampleArray(unsorted2);
cout << "Time taken: " << timeTaken.count() * 1000  << " milliseconds" << endl;

cout << endl <<"Unsorted Array 3:" << endl;
start = chrono :: high_resolution_clock :: now();
quickSort(unsorted3, 0, 300000 - 1);
stop = chrono :: high_resolution_clock :: now();
timeTaken = stop - start;
sampleArray(unsorted3);
cout << "Time taken: " << timeTaken.count() * 1000  << " milliseconds" << endl;

cout << endl << "Unsorted Array 4:" << endl;
start = chrono :: high_resolution_clock :: now();
quickSort(unsorted4, 0, 400000 - 1);
stop = chrono :: high_resolution_clock :: now();
timeTaken = stop - start;
sampleArray(unsorted4);
cout << "Time taken: " << timeTaken.count() * 1000  << " milliseconds" << endl;

cout <<  endl << "Unsorted Array 5:" << endl;
start = chrono :: high_resolution_clock :: now();
quickSort(unsorted5, 0, 500000 - 1);
stop = chrono :: high_resolution_clock :: now();
timeTaken = stop - start;
```

```
sampleArray(unsorted5);
cout << "Time taken: " << timeTaken.count() * 1000  << " milliseconds" << endl;

cout << "Sorted Arrays: " << endl;

cout <<  endl << "Sorted Array 1:" << endl;
start = chrono :: high_resolution_clock :: now();
quickSort(sorted1, 0, 100000 - 1);
stop = chrono :: high_resolution_clock :: now();
timeTaken = stop - start;
cout << "Time taken: " << timeTaken.count() * 1000  << " milliseconds" << endl;

cout << endl << "Sorted Array 2:" << endl;
start = chrono :: high_resolution_clock :: now();
quickSort(sorted2, 0, 200000 - 1);
stop = chrono :: high_resolution_clock :: now();
timeTaken = stop - start;
cout << "Time taken: " << timeTaken.count() * 1000  << " milliseconds" << endl;

cout << endl <<"Sorted Array 3:" << endl;
start = chrono :: high_resolution_clock :: now();
quickSort(sorted3, 0, 300000 - 1);
stop = chrono :: high_resolution_clock :: now();
timeTaken = stop - start;
cout << "Time taken: " << timeTaken.count() * 1000  << " milliseconds" << endl;

cout << endl << "Sorted Array 4:" << endl;
start = chrono :: high_resolution_clock :: now();
quickSort(sorted4, 0, 400000 - 1);
stop = chrono :: high_resolution_clock :: now();
timeTaken = stop - start;
cout << "Time taken: " << timeTaken.count() * 1000  << " milliseconds" << endl;

cout <<  endl << "Sorted Array 5:" << endl;
start = chrono :: high_resolution_clock :: now();
quickSort(sorted5, 0, 500000 - 1);
stop = chrono :: high_resolution_clock :: now();
timeTaken = stop - start;
```

```
cout << "Time taken: " << timeTaken.count() * 1000  << " milliseconds" << endl;


/*

   Middle Index

*/


// This round of sorts uses the middle as a pivot
cout << "\n\n\n\n Middle Pivot"<< endl;
populateRandoms(unsorted1, unsorted2, unsorted3, unsorted4, unsorted5);

cout <<  endl << " Unsorted Array 1:" << endl;
start = chrono :: high_resolution_clock :: now();
middleQuickSort(unsorted1, 0, 100000 - 1);
stop = chrono :: high_resolution_clock :: now();
timeTaken = stop - start;
sampleArray(unsorted1);
cout << "Time taken: " << timeTaken.count() * 1000  << " milliseconds" << endl;

cout << endl << "Unsorted Array 2:" << endl;
start = chrono :: high_resolution_clock :: now();
middleQuickSort(unsorted2, 0, 200000 - 1);
stop = chrono :: high_resolution_clock :: now();
timeTaken = stop - start;
sampleArray(unsorted2);
cout << "Time taken: " << timeTaken.count() * 1000  << " milliseconds" << endl;

cout << endl <<"Unsorted Array 3:" << endl;
start = chrono :: high_resolution_clock :: now();
middleQuickSort(unsorted3, 0, 300000 - 1);
stop = chrono :: high_resolution_clock :: now();
timeTaken = stop - start;
sampleArray(unsorted3);
cout << "Time taken: " << timeTaken.count() * 1000  << " milliseconds" << endl;
```

```
cout << endl << "Unsorted Array 4:" << endl;
start = chrono :: high_resolution_clock :: now();
middleQuickSort(unsorted4, 0, 400000 - 1);
stop = chrono :: high_resolution_clock :: now();
timeTaken = stop - start;
sampleArray(unsorted4);
cout << "Time taken: " << timeTaken.count() * 1000  << " milliseconds" << endl;

cout <<  endl << "Unsorted Array 5:" << endl;
start = chrono :: high_resolution_clock :: now();
middleQuickSort(unsorted5, 0, 500000 - 1);
stop = chrono :: high_resolution_clock :: now();
timeTaken = stop - start;
sampleArray(unsorted5);
cout << "Time taken: " << timeTaken.count() * 1000  << " milliseconds" << endl;

cout << "Sorted Arrays: " << endl;

cout <<  endl << "Sorted Array 1:" << endl;
start = chrono :: high_resolution_clock :: now();
middleQuickSort(sorted1, 0, 100000 - 1);
stop = chrono :: high_resolution_clock :: now();
timeTaken = stop - start;
sampleArray(sorted1);
cout << "Time taken: " << timeTaken.count() * 1000  << " milliseconds" << endl;

cout << endl << "Sorted Array 2:" << endl;
start = chrono :: high_resolution_clock :: now();
middleQuickSort(sorted2, 0, 200000 - 1);
stop = chrono :: high_resolution_clock :: now();
timeTaken = stop - start;
sampleArray(sorted2);
cout << "Time taken: " << timeTaken.count() * 1000  << " milliseconds" << endl;

cout << endl <<"Sorted Array 3:" << endl;
start = chrono :: high_resolution_clock :: now();
middleQuickSort(sorted3, 0, 300000 - 1);
stop = chrono :: high_resolution_clock :: now();
```

```
timeTaken = stop - start;
sampleArray(sorted3);
cout << "Time taken: " << timeTaken.count() * 1000  << " milliseconds" << endl;

cout << endl << "Sorted Array 4:" << endl;
start = chrono :: high_resolution_clock :: now();
middleQuickSort(sorted4, 0, 400000 - 1);
stop = chrono :: high_resolution_clock :: now();
timeTaken = stop - start;
sampleArray(sorted4);
cout << "Time taken: " << timeTaken.count() * 1000  << " milliseconds" << endl;

cout <<  endl << "Sorted Array 5:" << endl;
start = chrono :: high_resolution_clock :: now();
middleQuickSort(sorted5, 0, 500000 - 1);
stop = chrono :: high_resolution_clock :: now();
timeTaken = stop - start;
sampleArray(sorted5);
cout << "Time taken: " << timeTaken.count() * 1000  << " milliseconds" << endl;


/*

   Random Index

*/


//This round of sorts uses a random index as a pivot
cout << "\n\n\n\n Random Pivot"<< endl;
populateRandoms(unsorted1, unsorted2, unsorted3, unsorted4, unsorted5);

cout <<  endl << "Unsorted Array 1:" << endl;
start = chrono :: high_resolution_clock :: now();
randomQuickSort(unsorted1, 0, 100000 - 1);
stop = chrono :: high_resolution_clock :: now();
timeTaken = stop - start;
sampleArray(unsorted1);
```

```
cout << "Time taken: " << timeTaken.count() * 1000  << " milliseconds" << endl;

cout << endl << "Unsorted Array 2:" << endl;
start = chrono :: high_resolution_clock :: now();
randomQuickSort(unsorted2, 0, 200000 - 1);
stop = chrono :: high_resolution_clock :: now();
timeTaken = stop - start;
sampleArray(unsorted2);
cout << "Time taken: " << timeTaken.count() * 1000  << " milliseconds" << endl;

cout << endl <<"Unsorted Array 3:" << endl;
start = chrono :: high_resolution_clock :: now();
randomQuickSort(unsorted3, 0, 300000 - 1);
stop = chrono :: high_resolution_clock :: now();
timeTaken = stop - start;
sampleArray(unsorted3);
cout << "Time taken: " << timeTaken.count() * 1000  << " milliseconds" << endl;

cout << endl << "Unsorted Array 4:" << endl;
start = chrono :: high_resolution_clock :: now();
randomQuickSort(unsorted4, 0, 400000 - 1);
stop = chrono :: high_resolution_clock :: now();
timeTaken = stop - start;
sampleArray(unsorted4);
cout << "Time taken: " << timeTaken.count() * 1000  << " milliseconds" << endl;

cout <<  endl << "Unsorted Array 5:" << endl;
start = chrono :: high_resolution_clock :: now();
randomQuickSort(unsorted5, 0, 500000 - 1);
stop = chrono :: high_resolution_clock :: now();
timeTaken = stop - start;
sampleArray(unsorted5);
cout << "Time taken: " << timeTaken.count() * 1000  << " milliseconds" << endl;

cout << "Sorted Arrays: " << endl;

cout <<  endl << "Sorted Array 1:" << endl;
start = chrono :: high_resolution_clock :: now();
```

```
randomQuickSort(sorted1, 0, 100000 - 1);
stop = chrono :: high_resolution_clock :: now();
timeTaken = stop - start;
sampleArray(sorted1);
cout << "Time taken: " << timeTaken.count() * 1000  << " milliseconds" << endl;

cout << endl << "Sorted Array 2:" << endl;
start = chrono :: high_resolution_clock :: now();
randomQuickSort(sorted2, 0, 200000 - 1);
stop = chrono :: high_resolution_clock :: now();
timeTaken = stop - start;
sampleArray(sorted2);
cout << "Time taken: " << timeTaken.count() * 1000  << " milliseconds" << endl;

cout << endl <<"Sorted Array 3:" << endl;
start = chrono :: high_resolution_clock :: now();
randomQuickSort(sorted3, 0, 300000 - 1);
stop = chrono :: high_resolution_clock :: now();
timeTaken = stop - start;
sampleArray(sorted3);
cout << "Time taken: " << timeTaken.count() * 1000  << " milliseconds" << endl;

cout << endl << "Sorted Array 4:" << endl;
start = chrono :: high_resolution_clock :: now();
randomQuickSort(sorted4, 0, 400000 - 1);
stop = chrono :: high_resolution_clock :: now();
timeTaken = stop - start;
sampleArray(sorted4);
cout << "Time taken: " << timeTaken.count() * 1000  << " milliseconds" << endl;

cout <<  endl << "Sorted Array 5:" << endl;
start = chrono :: high_resolution_clock :: now();
randomQuickSort(sorted5, 0, 500000 - 1);
stop = chrono :: high_resolution_clock :: now();
timeTaken = stop - start;
sampleArray(sorted5);
cout << "Time taken: " << timeTaken.count() * 1000  << " milliseconds" << endl;
```

```
  // Clean up dynamic memory
  delete[] unsorted1;
  delete[] unsorted2;
  delete[] unsorted3;
  delete[] unsorted4;
  delete[] unsorted5;

  delete[] sorted1;
  delete[] sorted2;
  delete[] sorted3;
  delete[] sorted4;
  delete[] sorted5;

  return 0;
}
```

EnhancingQuickSort.h

```
#include <iostream>
using namespace std;

void sampleArray(int arr[]);
void populateRandoms(int arr1[], int arr2[], int arr3[], int arr4[], int arr5[]);
void populateSorted(int arr1[], int arr2[], int arr3[], int arr4[], int arr5[]);
void quickSort(int arr[], int  start, int end);
int partition(int arr[], int  start, int end);

int middlePartition(int arr[], int start, int end);
void middleQuickSort(int arr[], int start, int end);

int randomPartition(int arr[], int start, int end);
ele

void quickSort(int arr[], int start, int end){
  while (start < end) {
    if (end <= start) {
```

```
        return;
      }

      int pivot = partition(arr, start, end);

      if (pivot - start < end - pivot) {
        quickSort(arr, start, pivot - 1);
        start = pivot + 1;
      } else {
        quickSort(arr, pivot + 1, end);
        end = pivot - 1;
      }
    }
}

int partition(int arr[], int start, int end){
   int pivot = arr[end];
   int i = start - 1;

   for (int j = start; j <= end - 1; j++) {
      if (arr[j] < pivot) {
        i++;
        swap(arr[i], arr[j]);
      }
   }
   i++;
   swap(arr[i], arr[end]);
   return i;
}

void middleQuickSort(int arr[], int start, int end) {
   while (start < end) {
      // Partition the array and get pivot index
      int pivot = middlePartition(arr, start, end);

      // Recursively sort the smaller partition first
      if (pivot - start < end - pivot) {
        // Sort the left partition recursively
```

```
            middleQuickSort(arr, start, pivot - 1);
            // Update start to move to the larger right partition
            start = pivot + 1;
        } else {
            // Sort the right partition recursively
            middleQuickSort(arr, pivot + 1, end);
            // Update end to move to the larger left partition
            end = pivot - 1;
        }
    }
}

int middlePartition(int arr[], int start, int end) {
    // Calculate the middle index
    int mid = start + (end - start) / 2;
    // Swap the middle element with the last element to use it as pivot
    swap(arr[mid], arr[end]);
    int pivot = arr[end];  // Now, pivot is at the end
    int i = start - 1;

    for (int j = start; j <= end - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    // Place the pivot in its correct position
    i++;
    swap(arr[i], arr[end]);
    return i;
}

void randomQuickSort(int arr[], int start, int end) {
    while (start < end) {
        // Partition the array and get the pivot index
        int pivot = randomPartition(arr, start, end);

        // Sort the smaller partition first to minimize recursion depth
```

```
    if (pivot - start < end - pivot) {
      // Sort left partition recursively
      randomQuickSort(arr, start, pivot - 1);
      // Update start for the right partition
      start = pivot + 1;
    } else {
      // Sort right partition recursively
      randomQuickSort(arr, pivot + 1, end);
      // Update end for the left partition
      end = pivot - 1;
    }
  }
}

int randomPartition(int arr[], int start, int end) {
  // Select a random pivot index between start and end
  int randomIndex = start + rand() % (end - start + 1);

  // Swap the random pivot with the last element
  swap(arr[randomIndex], arr[end]);

  // Proceed with the normal partition process
  int pivot = arr[end];
  int i = start - 1;

  for (int j = start; j <= end - 1; j++) {
    if (arr[j] < pivot) {
      i++;
      swap(arr[i], arr[j]);
    }
  }

  // Swap the pivot element back to its correct position
  i++;
  swap(arr[i], arr[end]);
  return i;
}
```

```
void populateRandoms(int arr1[], int arr2[], int arr3[], int arr4[], int arr5[]){
   srand(time(0));
   for (int i = 0; i < 100000; i++){
      arr1[i] = (rand() % 1000000) + 1;
   }
   for (int i = 0; i < 200000; i++){
      arr2[i] = (rand() % 1000000) + 1;
   }
   for (int i = 0; i < 300000; i++){
      arr3[i] = (rand() % 1000000) + 1;
   }
   for (int i = 0; i < 400000; i++){
      arr4[i] = (rand() % 1000000) + 1;
   }
   for (int i = 0; i < 500000; i++){
      arr5[i] = (rand() % 1000000) + 1;
   }
}
void populateSorted(int arr1[], int arr2[], int arr3[], int arr4[], int arr5[]){
      for (int i = 0; i < 100000; i++){
      arr1[i] = i;
   }
   for (int i = 0; i < 200000; i++){
      arr2[i] = i;
   }
   for (int i = 0; i < 300000; i++){
      arr3[i] = i;
   }
   for (int i = 0; i < 400000; i++){
      arr4[i] = i;
   }
   for (int i = 0; i < 500000; i++){
      arr5[i] = i;
   }
}
void sampleArray(int arr[]){
   int i;
   int size = 25;
```

```
    for (i = 0; i < size; i+= 5)
        cout << arr[i] << ", ";

    cout << " ... ";

    for (i = 1000; i < 1000 + size; i+=5)
        cout << arr[i] << ", ";

    cout << " ... ";

    for (i = 2500; i < 2500 + size; i+=5)
        cout << arr[i] << ", ";
    cout << endl;
}
```