```
// Name: Connor Marler
// Instructor: Gerry Howser
// Date: 10/13/2022
// Project 2 Randomized Quick Sort Analysis


// Source: https://www.geeksforgeeks.org/quicksort-using-random-pivoting/

#include<iostream>
#include<vector>
#include <chrono>
#include <cassert>

using namespace std;


//Purpose: template function to find the position of pivot element
//PreCondition: takes in a templated function and the starting and ending index
//PostCondition: returns an interger of the pivot that is within the range of the
array
//Invariant: once within the if statement and after the swap, arr[j] <= arr[pivot]
template <typename T>
int Partition(T arr[], int start, int end){

      int pivot = end;
      int j = start;

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

      return j;

}


//Purpose: The purpose for this is to randomize the partition in order to mitigate
risk
//PreCondition: takes in an array (arr) and two intergers, one is the lower bound
and one
//              is the higher bound
//PostCondition: reurns a recursive call of the Partition() function
template <typename T>
int Randomized_Partition(T arr[], int low, int high)
{
    // Generate a random number in between
    // low .. high
    srand(time(NULL));
    int random = low + rand() % (high - low);

    // Swap A[random] with A[high]
    swap(arr[random], arr[high]);

    return Partition(arr, low, high);
```

```
}

//Purpose: template function to perform quick sort on array arr
//PreCondition: takes in a templated function and the starting and ending index
//PostCondition: the array is sorted in ascending order
template <typename T>
void Randomized_Quicksort(T arr[], int start, int end ){

      if(start<end){
            int p = Randomized_Partition(arr,start,end);
            Randomized_Quicksort(arr,start,p-1);
            Randomized_Quicksort(arr,p+1,end);
      }
}

//Purpose: Template function to print array
//PreCondition: takes in an array and a value of the size of the array
//PostCondition: outputs the array givens
template<typename T>
void PrintArray(T arr[], int n)
{
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";
    cout << "\n\n";
}


//Purpose: take in an array and fill it with a predetermined element set
//PreCondition: must receive arr[size] and a string value determining how to fill
the set
//PostCondition: must fill arr[size] with the values in the appropriate as
determined by preSortOrder
void populateArray(int *arr, const int size, const string preSortOrder)
{

  // This if else tree could also be re-written as a switch case
  if(preSortOrder == "assPreSortOrder")
  {
    for(int i = 0; i <= size; i++)
      {
        arr[i] = i + 1;
      }
  }
  else if(preSortOrder == "decPreSortOrder")
  {
    for(int j = 0; j <= size; j++)
      {
        arr[j] = size - j;
      }
  }
  else if(preSortOrder == "randPreSortOrder")
  {
    for(int k = 0; k <= size; k++)
      {
        arr[k] = rand() % size + 1;
      }
  }
  else // error handling
  {
```

```
    cout << "ERROR: Something broke. Please quit and try again." << endl;
  }
}

int main()
{

  // These are what we will use for the time measurement
  using chrono::high_resolution_clock;
  using chrono::duration_cast;
  using chrono::duration;
  using chrono::nanoseconds;

  srand (time(NULL));
  const int SIZE = 500;
  string preSortOrder;
  int innerLoop = 1000;
  int outerLoop = 10;
  int averageTime = 0;
  int time = 0;


  int mainArray[SIZE]; // This the hard, original copy of the array
  int workingArray[SIZE]; // This is the copy used to work with and innerLoop
through


  cout << "RANDOMIZED QUICKSORT AVG TIMES" << endl;
  for(int i = 0; i <= 2; i ++) // loops through each case
  {
    if(i == 0)
    {
      preSortOrder = "assPreSortOrder";
      cout << endl << "Ascending Order Time:" << endl;
    }
    else if(i == 1)
    {
      preSortOrder = "decPreSortOrder";
      cout << endl << "Descending Order Time:" << endl;
    }
    else if(i == 2)
    {
      preSortOrder = "randPreSortOrder";
      cout << endl << "Random Order Time:" << endl;
    }

    populateArray(mainArray, SIZE, preSortOrder);
    averageTime = 0;

    for(int k = 0; k < outerLoop; k++) // takes 10 data point
    {
      averageTime = 0;
      for(int j = 0; j < innerLoop; j++) // perform the sort 1000 times and take an
average for one data point
      {
        for (int a = 0; a < SIZE; a++) // This copies over the original array onto
a working copy
        {
          workingArray[a] = mainArray[a];
```

```
        }
        //cout << "Before Sort" << endl;
        //PrintArray(workingArray, SIZE);
        auto time1 = high_resolution_clock::now(); // take initial time
        Randomized_Quicksort(workingArray, 0, SIZE - 1);  // do the sort
        auto time2 = high_resolution_clock::now(); // time the after time
        //cout << "After Sort" << endl;
        //PrintArray(workingArray, SIZE);
        auto nanoSeconds = duration_cast<nanoseconds>(time2 - time1); // find the
difference of the times
        time = nanoSeconds.count();  // convert to an int
        averageTime += time;
      }
      averageTime /= innerLoop;
      cout << "Trial # " << k + 1 << ": " << averageTime << " ns" << endl;
    }
  }
}
```