LAB 5: ALGORITHMS COMPARED BY:AMULDEEP DHILLON

Algorithms

We will be creating a program that runs multiple algorithms that sort the same array of integers. It will time how long it takes for each sorting allowing the user to see which is faster.

We will be comparing Mergesort and Quicksort

Quicksort header file

```
class Quicksort{
private:
    int split(int [],int);
public:
    void sort(int[],int); //array and size
};
```

Contains the Quicksort class

Mergesort header file

```
class Mergesort{
private:
    int* work;
    void merge(int a1[], int n1, int a2[], int n2);
public:
    Mergesort (int n) {work = new int[n];};
    ~Mergesort () {delete [] work;};
    void sort(int[],int); //array and size
};
```

Contains the Mergesort class

Merge Sort Algorithm

```
#include "Mergesort.h"
void swap(int& a, int& b){a = a^b; b = a^b; a = a^b;}
void Mergesort::merge (int a1[], int n1, int a2[], int n2){
    int i1=0,i2=0;
    for(int i = 0; i < n1+n2; i++){
        if(i1 < n1 and i2 < n2)
        {if(a1[i1] < a2[i2]) {work[i] = a1[i1]; i1++;}
        else {work[i] = a2[i2]; i2++;}}
        else if(i1< n1) {work[i] = a1[i1]; i1++;}
        else{work[i] = a2[i2]; i2++;}}
    for(int i = 0; i < n1+n2; i++)
        a1[i] = work[i];}
void Mergesort::sort(int a[],int n){int n1, n2; int* a2;
    n1 = n/2;  n2 = n - n1; a2 = &a[n1];
    //base case
    if(n <= 2){
        if((n == 2) && (a[1] < a[0]))
        swap(a[0],a[1]);}
    else{sort(a,n1);   //sort the left half
        sort(a2,n2); //sort the right half
        merge(a,n1,a2,n2);}}
```

- seperates array into pieces of either one or two elements then sorts those individual pieces
- combines the sorted pieces into, eventually, one array

Quick Sort Algorithm

```
#include "Quicksort.h"
extern void swap(int &a, int &b);
int Quicksort::split(int a[],int n)
{   int p = n/2;
    int pivot = a[p];
    int i =0;
    int j=n-1;
    while (i<= j)
      if(a[i] <= pivot) i++;
      else if(a[j] >= pivot) j--;
      else {swap(a[i],a[j]); i++;j--;}
    if(p<j) {swap(a[p],a[j]); p=j;}
    else if(p>i) {swap (a[p],a[i]); p=i;}
    return p;}
void Quicksort::sort(int a[],int n){
    if(n<=2)
    {
      if(n==2 and a[1]<a[0]) swap(a[0],a[1]);
  }else {
    int p = split(a,n);   //find a pivot
    sort(a,p);   //sorting the left subsection of a
    sort(&a[p+1], n-p-1);
    } //sorting the right subsection
}
```

- picks an elements and sorts all based on the element
- continually repeat for each side of the original element till eventually there are only one element on either side therefore the array is organized

Main

```
#include <iostream>
#include <chrono>
using namespace std::chrono;
#include "Quicksort.h"
#include "Mergesort.h"
const int N = 1000000;const int MAX = 10000; int original[N]; int a[N];
void copy(int a[], int b[]){
    for(int i = 0; i < N; i++) b[i] = a[i];}
int main(){srand(time(0)); for(auto &e : original) e = rand() % MAX;
    copy(original,a); auto t0 = high_resolution_clock::now();
    auto t1 = high_resolution_clock::now(); copy(original,a);
    Quicksort q;t0 = high_resolution_clock::now();
    q.sort(a,N);t1 = high_resolution_clock::now();
    std::cout << "Quick Sort: " << duration_cast<nanoseconds>(t1-t0).count() <<" Nanoseconds" << std::endl;
    //for(auto e:a) std::cout << e << "\t"; //print array
    std::cout << std::endl; copy(original,a); Mergesort m(N);
    t0 = high_resolution_clock::now();
    m.sort(a,N);
    t1 = high_resolution_clock::now();
    std::cout << "Merge Sort: " << duration_cast<nanoseconds>(t1-t0).count() <<" Nanoseconds" << std::endl;
    //for(auto e:a) std::cout << e << "\t"; //print array
    std::cout << std::endl;
}
```

- records time
- run algorithm
- record second time
- calculate difference
- display difference and sorted array(commented out since it takes too long to display one million different elements in an array)
- repeat for second algorithm

```
debian@debian:~/cs124/lab5$ ./lab
Quick Sort: 1541 Nanoseconds
1276    2154    3874    6529    7110
Merge Sort: 1956 Nanoseconds
1276    2154    3874    6529    7110
```

These are just two out of many tests but the pattern is the same Quick sort is reliably faster.

```
debian@debian:~/cs124/lab5$ ./lab
Quick Sort: 2833 Nanoseconds
3153    3514    5756    6665    9511
Merge Sort: 3323 Nanoseconds
3153    3514    5756    6665    9511
```

```
debian@debian:~/cs124/lab5$ ./lab
Quick Sort: 947708809 Nanoseconds

Merge Sort: 773497343 Nanoseconds
```

When the number of elements is increased, to 1 million in this case, the difference between the two algorithms increases, but the algorithm that is faster most of the time is Mergesort as opposed to the previous test with only 5 values it was Quicksort.

```
debian@debian:~/cs124/lab5$ ./lab
Quick Sort: 980544125 Nanoseconds

Merge Sort: 728343814 Nanoseconds
```

Analysis pt.2

Even though Merge sort and Quick sort are both Big O of (nLog(n)) the reason Mergesort is faster lies in the Math of Big O. The equation for the time using big O of nLog(n) is Time = (number of Loops)(Logbase2(number of steps))(Time per step). So to compare two alorithms of type nLog(n) you simply have to look at the number of loops and steps, since the time per step is almost impossible to calculate reliably. That is where Merge Sort shines, it has less numbers of steps per loop so it has a overall lower number to multiply with the other two, thus making it faster. Another important point to make is that Quick sort performs better when it picks a better "pivot" point, so it's chance of success is random, while Merge sort just needs elements to already be in order, or atleast partially in order to perform better, something more likely to happen when there are more values to begin with. More elements are not useful when all that matters is where a randomly picked value lies on the scale.

It is important to note that these calculations do not account for best or worst cases, only the average case.
In fact, Merge sort is always of Big O nLog(n) for best, worst, or average case. It is Quick sort that changes and not for the better, since Quick Sort's best and average cases are nLog(n) but it's worst case is n squared, worse than nLog(n). However, it is important to add that Merge sort requires alot more memory since the array has to basically be copied while quick sort can just rely on the memory the array already takes up.