Analysis of Algorithms 2021/2022

Practice 2

Pablo Almarza, Group.

Code	Plots	Memory	Total

1. Introduction.

In this practice some of the functions created in the previous practice will be used and MergeSort and QuickSort will be created, two new sorting algorithms.

2. Objectives

2.1 Section 1

Implement the code for a working MergeSort algorithm. This algorithm will be implemented recursively. Also changing the file exercise4 to check the correctness of the algorithm

2.2 Section 2

The goal for this exercise is to deal with the problems this new algorithm may cause when implementing it in the exercise5 file to show the average time, the minimum, maximum and average basic operations in relation to the size of the array to order.

2.3 Section 3

In this exercise the QuickSort algorithm is created, making it work with two additional functions: split, to move the pivot into the correct position, and median, to select the pivot (always the first element of the table in this case).

2.4 Section 4

In this exercise the functions created in times.c will be used like in the last practice to obtain the maximum, minimum and average number of basic operations, and the average clock time depending on the size of the permutation.

2.5 Section 5

In this exercise a new quicksort function will be created, deleting the tail recursion.

3 Tools and Methodology

It has been using Visual Studio on Linux. Using C/C++ extension for VS and Valgrind when executing the programs to check errors and memory leaks.

3.1 Section 1

For exercise 1 it was asked to implement the merge sort recursively following the pseudocode in moodle. This brought problems like the counting of the basic operations. Regarding the workspace, only the terminal was necessary to obtain the results.

3.2 Section 2

In exercise 2 GNUplot was used again to generate the plots asked (times and basic operations plots).

3.3 Section 3

For this exercise again the pseudocode posted on Moodle of the quicksort was followed. However, a modification was necessary in order to implement the position of the pivot as a pointer moving across functions. Also we had to move this pivot to the first position everytime split function was called in order for it to work.

3.4 Section 4

The only thing to do was to change the function used in exercise5.c to quicksort.

3.5 Section 5

Here, the new function quicksort_ntr was created to eliminate tail recursion. It uses a while loop with first<last condition. So that when first = last the function would not enter the loop and return 1 (as the count is initialized to 1) as the base case. That way tail recursion is gone.

4. Source code

4.1 Section 1

```
int merge(int* tabla, int ip, int iu, int medio) {

 int i, j, k, count = 0, lowValue = medio - ip + 1, upValue = iu -
medio;

 int *lowerTab, *upperTab;

  lowerTab = (int*) malloc(lowValue * sizeof(int));

  if(!lowerTab) return ERR;

  upperTab = (int*) malloc(upValue * sizeof(int));

  if(!upperTab) {

   free(lowerTab);

   return ERR;

 }

  for (i = 0; i < lowValue; i++) {

   lowerTab[i] = tabla[ip + i];

 }

 for (j = 0; j < upValue; j++) {

   upperTab[j] = tabla[medio + 1 + j];

 }

 i = 0;

 j = 0;

 k = ip;
```

```
/*Perform the key comparison, raise the count by one, and merge
the table */

while (i < lowValue && j < upValue) {

  count++;

  if (lowerTab[i] <= upperTab[j]) {

    tabla[k] = lowerTab[i];

    i++;

  } else {

    tabla[k] = upperTab[j];

    j++;

  }

  k++;

}

/* Copy the remaining elements of lowerTab[], if there are any */

while (i < lowValue) {

  tabla[k] = lowerTab[i];

  i++;

  k++;

}

free(lowerTab);

 /* Copy the remaining elements of upperTab[], if there are any */

while (j < upValue) {

  tabla[k] = upperTab[j];
```

```
    j++;
    k++;
  }
  free(upperTab);
   return count;
}
int MergeSort(int* tabla, int ip, int iu) {
  int medio = ip + (iu - ip) / 2, auxCount = 0, count = 0;
  if (!tabla || ip >= iu) {
    return ERR;
  }
  auxCount = MergeSort(tabla, ip, medio);
  if (auxCount > 0) {
    count = auxCount;
  } else {
    count = 0;
  }
  auxCount = MergeSort(tabla, medio + 1, iu);
  if (auxCount > 0) {
    count += auxCount;
  } else {
    count += 0;
```

```
 }

 return count + merge(tabla, ip, iu, medio);

}
```

4.3 Section 3

```
int median(int *tabla, int ip, int iu,int *pos) {

 if (!tabla || ip > iu || !pos) return ERR;

 *pos = ip;

 return 0;

}

int split(int* tabla, int ip, int iu,int *pos) {

 int i, k, count;

 if (!tabla || ip > iu || ip < 0 || !pos) return ERR;

 count = median(tabla, ip, iu, pos);

  if((*pos) > iu || (*pos) < ip){

    return count;

 }

 k = tabla[*pos];

 swap(&tabla[ip], &tabla[*pos]);

 *pos = ip;

 for (i = ip + 1; i <= iu; i++) {

    count++;
```

```
    if (tabla[i] < k) {
      (*pos)++;
      swap(&tabla[i], &tabla[*pos]);
    }
  }
  swap(&tabla[ip], &tabla[*pos]);
  return count;
}
int QuickSort(int* tabla, int ip, int iu) {
  int pos;
  int count = 0;
  int auxCount;
  if(!tabla || ip >= iu) return ERR;
  auxCount = split(tabla, ip, iu, &pos);
  if(auxCount == ERR) return ERR;
  count = auxCount;
  auxCount = QuickSort(tabla, pos+1, iu);
  if (auxCount > 0) {
    count += auxCount;
  } else {
    count += 0;
  }
```

```
 auxCount = QuickSort(tabla, ip, pos-1);

 if (auxCount > 0) {

   count += auxCount;

 } else {

   count += 0;

 }

 return count;

}
```

4.5 Section 5

```
int QuickSort_ntr(int* table, int ip, int iu) {

 int pos = 1, count = 1;



 if (table == NULL || ip < 0 || iu < ip) {

   return ERR;

 }

 while (ip < iu) {

   count += split(table, ip, iu, &pos);

   if (count == ERR)return ERR;

   if(pos - ip < iu - pos) {

     QuickSort_ntr(table, ip, pos - 1);

     ip = pos + 1;
```

```
   } else {

     QuickSort_ntr(table, ip, iu);

     iu = pos - 1;

   }

 }

 return count;

}
```

5. Results, Plots

5.1 Section 1

./exercise4 -size 20

Practice number 1, section 4

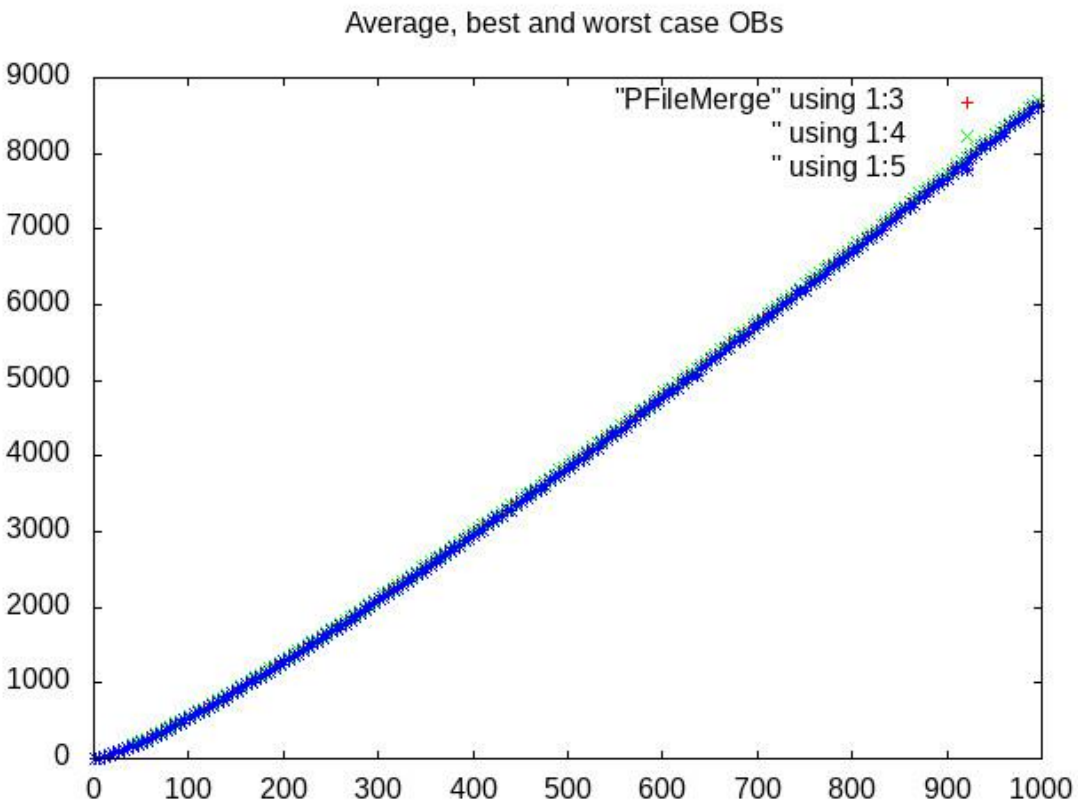
Done by: Pablo Almarza

Group: Your group

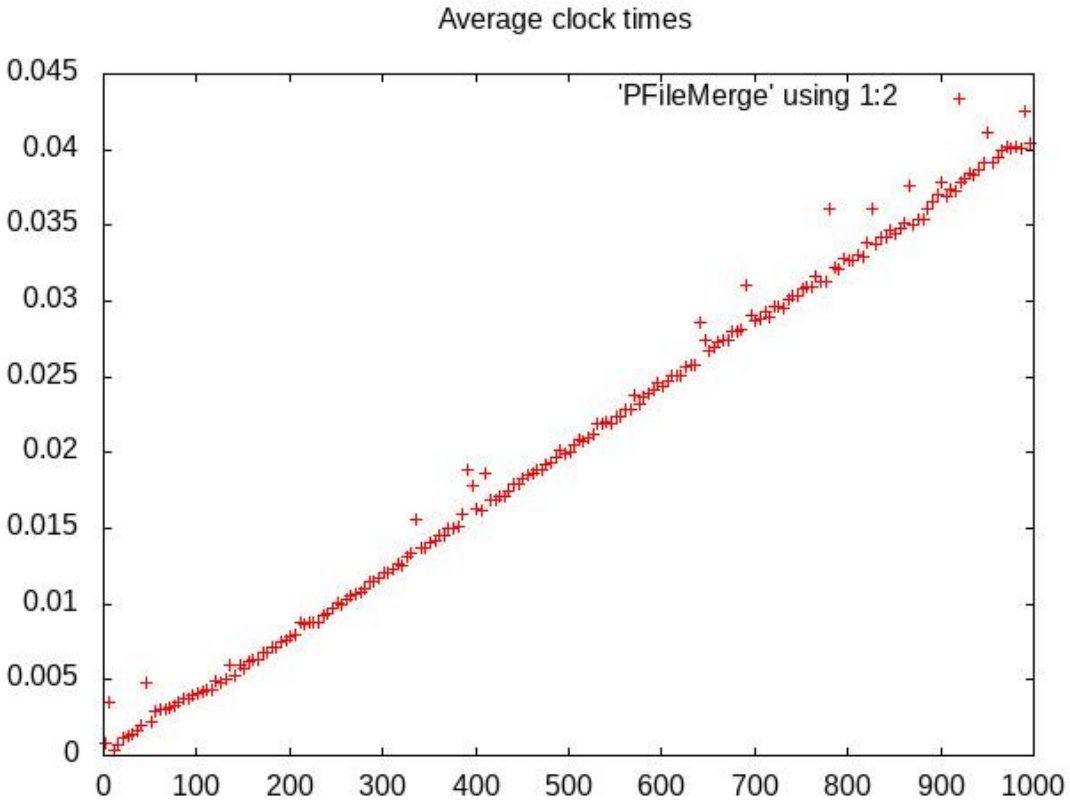
1 2 3 4 5 6 7 8 9 10 11
12 13 14 15 16 17 18 19 20

69

5.2 Section 2



Since MergeSort has almost the same number of OBs for the best and the worst case, there is no much difference in them.



MergeSort time is very optimal no matter how big the array is.

5.3 Section 3

./exercise4 -size 20

Practice number 1, section 4

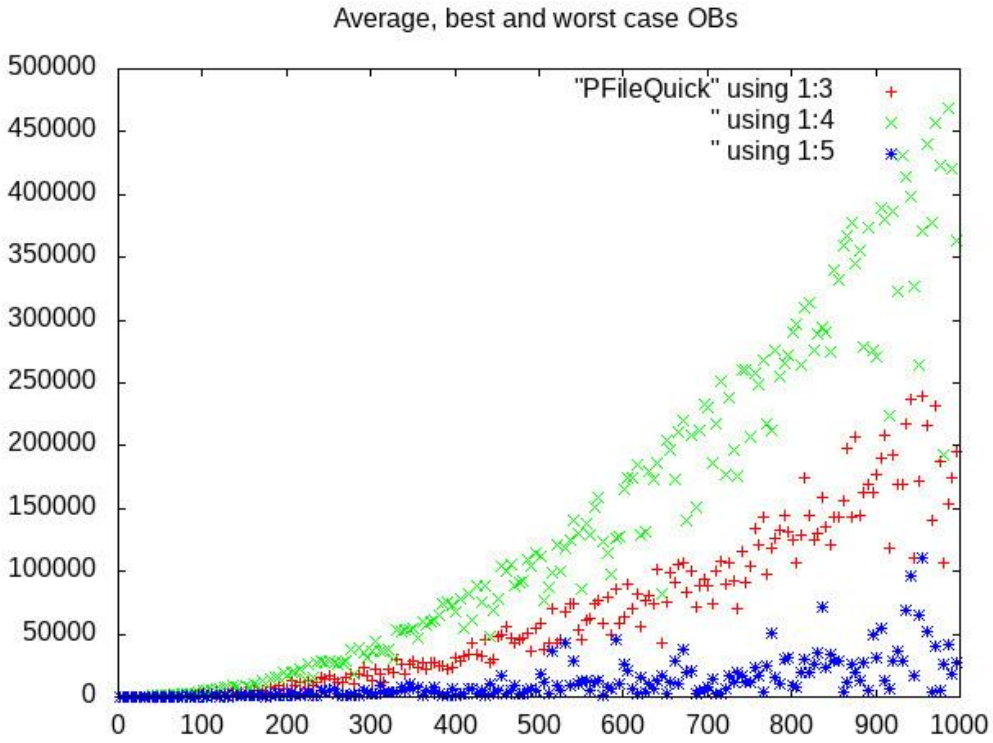
Done by: Pablo Almarza

Group: Your group

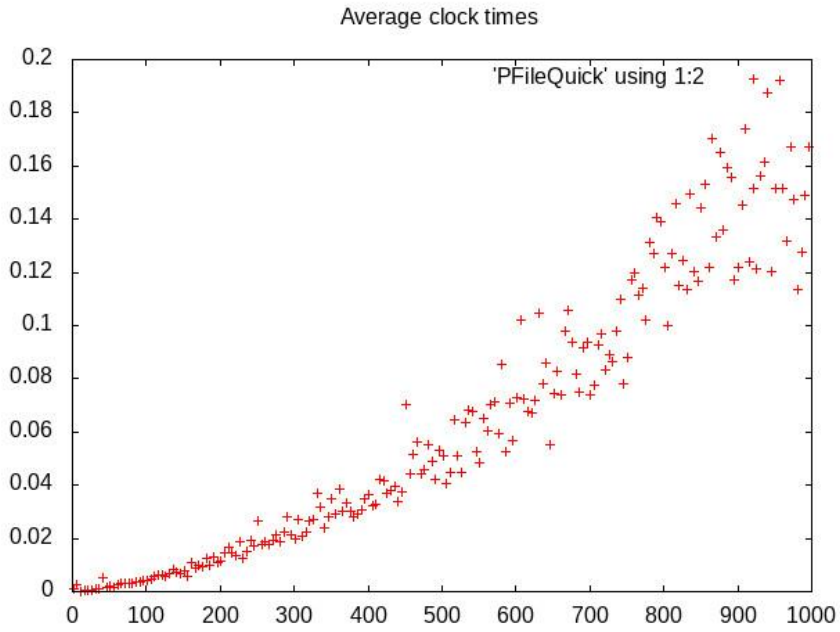
1 2 3 4 5 6 7 8 9 10 11
12 13 14 15 16 17 18 19 20

72

5.4 Section 4

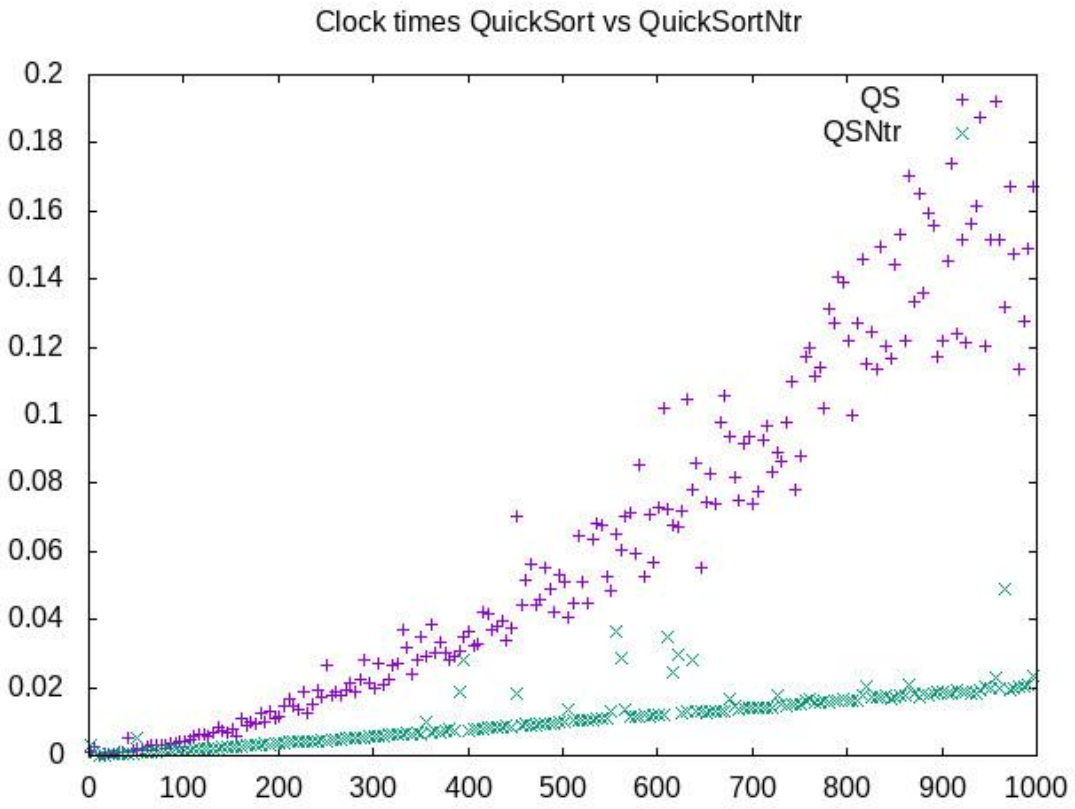


We can see that the Obs are very different since the worst case for QuickSort is $O(N^2)$ while the best case is 2NlogN. Also we can see that the OBs are much more than MergeSort.



We can see that the time QuickSort needs is not too much with short arrays but the larger the size is, the more time it takes to end. Worse than MS when the array is big.

5.5 Section 5



We can observe that QuickSort_ntr is much better than the original one.

5. Answers to theoretical Questions.

5.1 Question 1

Both algorithms follow the theoretical lines. But the merge sort seems to follow the theoretical line much more sharply. This is because the merge sort is a very efficient sorting algorithm and the random factor won't affect it as much as other sorting methods.

5.2 Question 2

It can be observed that QuickSort without tail recursion has a much better worst case than the original QuickSort. In fact we can see that the worst case for QuickSort_ntr is O(NlogN) while the original QuickSort is $O(N^2)$.

5.3 Question 3

$W_{QS}(N) = O(N^2)$

$B_{QS}(N) = 2Nlog(N) + O(N)$

$W_{MS}(N) = Nlog(N) + O(N)$

$B_{MS}(N) = (½)Nlog(N)$

To calculate the best case for the MergeSort, it would be when the smaller numbers are grouped in the left half so the bigger mergings don't take too long to merge. However the merge sort doesn't seem affected as much as other sorting algorithms by the numbers not being ordered. For the QuickSort algorithm the best case scenario is that the pivot is the middle value of the table. To calculate each case the median function could be modified to search for the intermediate value pivot for the best case, the lower or highest value pivot for the worst case, and the average between the intermediate and lower or highest value pivot for the average case.

5.4 Question 4

MergeSort is more consistent and more efficient for larger arrays than QuickSort, because of the worst case theoretical formulas. However, QuickSort is an internal sorting algorithm, so it requires very little memory management, contrary to the merge which allocates auxiliary arrays.

6. Final Conclusions.

There are several conclusions in this practice. First of all, the way of sorting, while MergeSort is more intuitive since it always divides the array in 2 halves, in QuickSort there are many possibilities which makes it more difficult to follow.

Another thing is the worst case complexity, while MergeSort have the same complexity for worst case and average case (O(Nlog(N)), QuickSort worst case is equal to $O(N^2)$, that means that if we have the worst case in this second algorithm, it will need a lot more comparisons than QuickSort Furthermore, QuickSort is not good with large arrays but MergeSort is.

A positive thing for QuickSort would be the space requirement because it doesn't require any additional storage, but MergeSort does, in order to store the auxiliary arrays. Not only that, because MergeSort needs even more auxiliary memory for sorting, since it is a external sorting method while QuickSort doesn't require that,