Analysis of Algorithms 2020/2021

Practice 2

S. Xiao Fernandez Marin

Code	Plots	Memory	Total

1. Introduction.

In this practice, starting from the time functions I had coded in the previous one, I study the efficiency of two new algorithms which use divide and conquer approach. They are the functions: MergeSort and QuickSort. For each algorithm, it will be analyzed on tables of different sizes and also the results obtained will be compared with the theoretical study of the algorithm.

2. Objectives

2.1 Section 1

First of all, I studied the sorting algorithm corresponding to the method MergeSort. This algorithm works dividing each table in subtables until it has each element individually, when it starts sorting by comparing each element and combining them.

2.2 Section 2

In this part, I proved that the algorithm MergeSort works correctly, by adapting exercise5.c, given in the previous practice. It is expected that the average clock time it takes to sort is smaller than the one in InsertSort, as MergeSort does not work with the table at all once, but dividing and working with smaller elements.

2.3 Section 3

In this section I have implemented the sorting algorithm QuickSort. This algorithm consists of three functions: quicksort, split and median.

2.4 Section 4

The objective of this exercise is to study the execution of QuickSort algorithm by adapting exercise5.c, and then comparing it with the theoretical result of the algorithm.

2.5 Section 5

In this exercise, I had to add the function quicksort_ntr. This function is the quicksort function but without the tail recursion.

3 Tools and Methodology

Basically, the tool that I have been using during this task is Visual Studio Code, because it facilitates us the debugging of the code. I have also executed all the exercises with valgrind to prove that there were no dynamic memory allocation errors. To create the plots, I have used Gnuplot.

3.1 Section 1

To do this exercise, I have guided myself using the pseudocode we can find in the slides of the subject. First I understood how the MergeSort algorithm works writing it in a paper. Then, I started to code using the pseudocode. Lastly, to prove that it sorts correctly, I have used the exercise4.c given in the previous assignment and I have also executed it with valgrind to see there were no memory errors.

3.2 Section 2

Then, I have adapted the file exercise5.c of the previous practice for using MergeSort algorithm, and I have obtained the table with the average clock time, and the average, minimum and maximum number of times that the BO is executed depending on the size of the permutation. In this part I have also plotted the resulting values to compare them with the theoretical result.

3.3 Section 3

For this exercise, I have also used the slides and the pseudocode from the second unit of the subject to understand how the algorithm QuickSort works and then to code it. This algorithm requires two additional functions: split, that divides the elements of the table, and median, that determines the position of the pivot. Then, I have also used the exercise4.c adapted to QuickSort to check that it orders correctly.

3.4 Section 4

In this exercise, again adapted the file exercise5.c of the previous practice for using QuickSort algorithm, I have obtained the table with the average clock time, and the average, minimum and maximum number of times that the BO is executed depending on the size of the permutation. Then, as in exercise 2, I have plotted the resulting values and compared them with the theoretical result.

3.5 Section 5

We were asked to implement a new functions to remove the tail recursion. Using the file exercise5.c, I compared the average clock time, and the average, minimum and maximum number of times that the BO is executed with each function implemented.

4. Source code

4.1 Section 1

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

  int middle = 0, i = 0;
  int aux1 = 0, aux2 = 0, ob = 0;

  if (table == NULL || ip < 0 || ip > iu)return ERR;

  if (ip == iu)return OK;

  else{
    middle = (int)(iu + ip) / 2;

    aux1 = mergesort(table, ip, middle);
    if (aux1 == ERR) return ERR;

    else ob = ob + aux1;

    aux2 = mergesort(table, middle + 1, iu);
    if (aux2 == ERR) return ERR;

    else ob = ob + aux2;

    i = merge(table, ip, iu, middle);
    if (i == ERR) return ERR;
    i = i + ob;
  }
  return i;

}
```

```
int merge(int* table, int ip, int iu, int imiddle){


  int i = ip, j = imiddle + 1, k = 0, cont = 0;
  int *auxtable = NULL;

  if (table == NULL || ip < 0 || iu < ip || imiddle > iu || imiddle < ip) return ERR;

  auxtable = (int *)malloc(((iu - ip + 1) * sizeof(int)));
  if (auxtable == NULL) return ERR;

  while (i <= imiddle && j <= iu) {
    cont++;
    if (table[i] < table[j]){
      auxtable[k] = table[i];
      i++;
    }
    else{
      auxtable[k] = table[j];
      j++;
    }
    k++;
  }

  if (i > imiddle){
    while (j <= iu){
      auxtable[k] = table[j];
      j++;
      k++;
    }
  }
  else if (j > iu){
    while (i <= imiddle){
      auxtable[k] = table[i];
      i++;
      k++;
    }
  }

  for (i = ip, j = 0; i <= iu; i++, j++){
    table[i] = auxtable[j];
  }

  free(auxtable);

  return cont;

}
```

4.3 Section 3

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

  int pos = 1, middle = 0, ob = 0, aux_left = 0, aux_right = 0;

  if (!table || ip < 0 || iu < 0 || iu < ip) return ERR;

  if (ip == iu) return ob;

  ob += split(table, ip, iu, &pos);
  if (ob == ERR)return ERR;

  middle = pos;

  if (ip < middle - 1) {
    aux_left = quicksort(table, ip, middle - 1);
    if (aux_left == ERR)return ERR;
    ob = ob + aux_left;
  }

  if (middle + 1 < iu) {
    aux_right = quicksort(table, middle + 1, iu);
    if (aux_right == ERR) return ERR;
    ob = ob + aux_right;
  }

  return ob;

}

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

  int i = 0, k = 0, ob = 0, swap = 0, swap1 = 0, swap2 = 0;

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

  ob += median(table, ip, iu, pos);
  if (ob == ERR) return ERR;

  if ((*pos) > iu || (*pos) < ip) return ERR;

  k = table[*pos];

  swap = table[ip];
  table[ip] = table[*pos];
  table[*pos] = swap;

  *pos = ip;

  for (i = ip + 1; i <= iu; i++){
    if (table[i] < k){
      (*pos)++;
      swap1 = table[i];
      table[i] = table[*pos];
      table[*pos] = swap1;
    }
    ob++;
  }

  swap2 = table[ip];
  table[ip] = table[*pos];
  table[*pos] = swap2;

  return ob;

}
```

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

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

  *pos = ip;

  return 0;

}
```

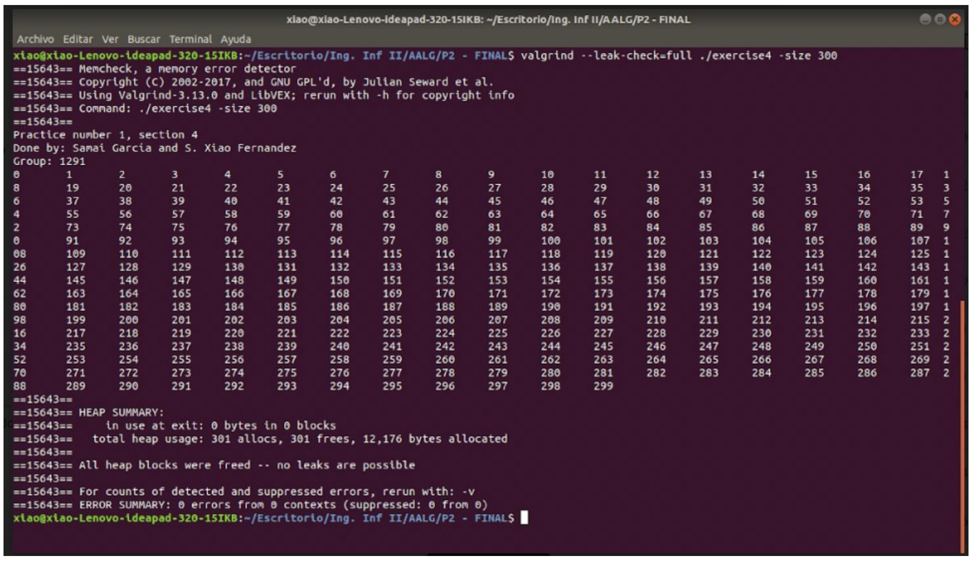
4.5 Section 5

```
int quicksort_ntr (int* table, int ip, int iu){
 int pos = 1, middle = 0, ob = 0, aux_left = 0;

  if (!table || ip < 0 || iu < 0 || iu < ip) return ERR;

  while (ip < iu) {

    ob += split(table, ip, iu, &pos);
    if (ob == ERR)return ERR;

    middle = pos;

    aux_left = quicksort_ntr(table, ip, middle);
    if (aux_left == ERR)return ERR;
    ob = ob + aux_left;

    ip = middle + 1;
  }

  return ob;

}
```

5. Results, Plots

5.1 Section 1

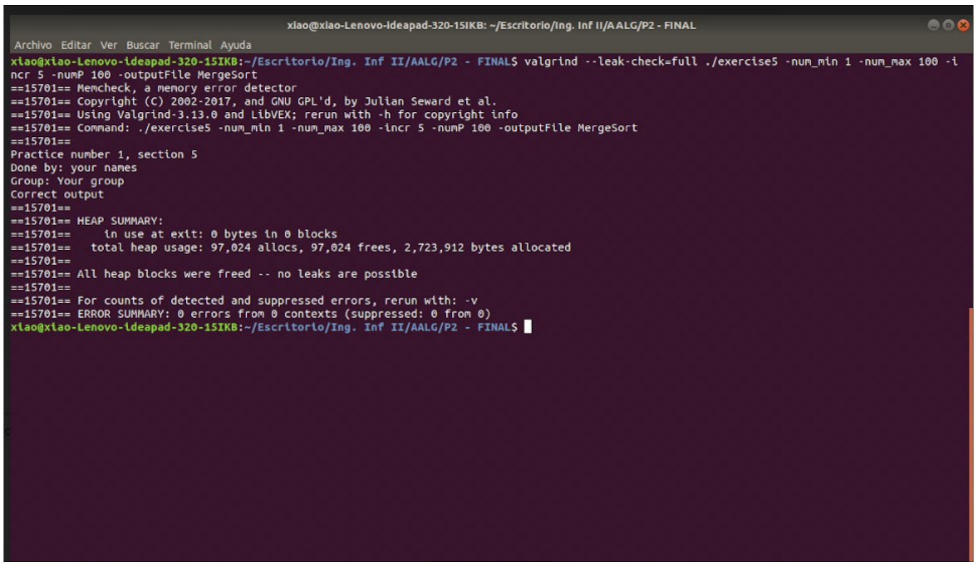
I executed this exercise with exercise4.c and with valgrind to prove that the algorithm MergeSort works correctly and that there are not memory errors. We also checked that it orders correctly by entering a table of size 300.



```
xiao@xiao-Lenovo-ideapad-320-15IKB:~/Escritorio/Ing. Inf II/AALG/P2 - FINAL$ valgrind --leak-check=full ./exercise4 -size 300
==15643== Memcheck, a memory error detector
==15643== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==15643== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==15643== Command: ./exercise4 -size 300
==15643==
Practice number 1, section 4
Done by: Samai Garcia and S. Xiao Fernandez
Group: 1291
0       1       2       3       4       5       6       7       8       9       10      11      12      13      14      15      16      17      1
8       19      20      21      22      23      24      25      26      27      28      29      30      31      32      33      34      35      3
6       37      38      39      40      41      42      43      44      45      46      47      48      49      50      51      52      53      5
4       55      56      57      58      59      60      61      62      63      64      65      66      67      68      69      70      71      7
2       73      74      75      76      77      78      79      80      81      82      83      84      85      86      87      88      89      9
0       91      92      93      94      95      96      97      98      99      100     101     102     103     104     105     106     107     1
08      109     110     111     112     113     114     115     116     117     118     119     120     121     122     123     124     125     1
26      127     128     129     130     131     132     133     134     135     136     137     138     139     140     141     142     143     1
44      145     146     147     148     149     150     151     152     153     154     155     156     157     158     159     160     161     1
62      163     164     165     166     167     168     169     170     171     172     173     174     175     176     177     178     179     1
80      181     182     183     184     185     186     187     188     189     190     191     192     193     194     195     196     197     1
98      199     200     201     202     203     204     205     206     207     208     209     210     211     212     213     214     215     2
16      217     218     219     220     221     222     223     224     225     226     227     228     229     230     231     232     233     2
34      235     236     237     238     239     240     241     242     243     244     245     246     247     248     249     250     251     2
52      253     254     255     256     257     258     259     260     261     262     263     264     265     266     267     268     269     2
70      271     272     273     274     275     276     277     278     279     280     281     282     283     284     285     286     287     2
88      289     290     291     292     293     294     295     296     297     298     299
==15643==
==15643== HEAP SUMMARY:
==15643==     in use at exit: 0 bytes in 0 blocks
==15643==   total heap usage: 301 allocs, 301 frees, 12,176 bytes allocated
==15643==
==15643== All heap blocks were freed -- no leaks are possible
==15643==
==15643== For counts of detected and suppressed errors, rerun with: -v
==15643== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
xiao@xiao-Lenovo-ideapad-320-15IKB:~/Escritorio/Ing. Inf II/AALG/P2 - FINAL$
```

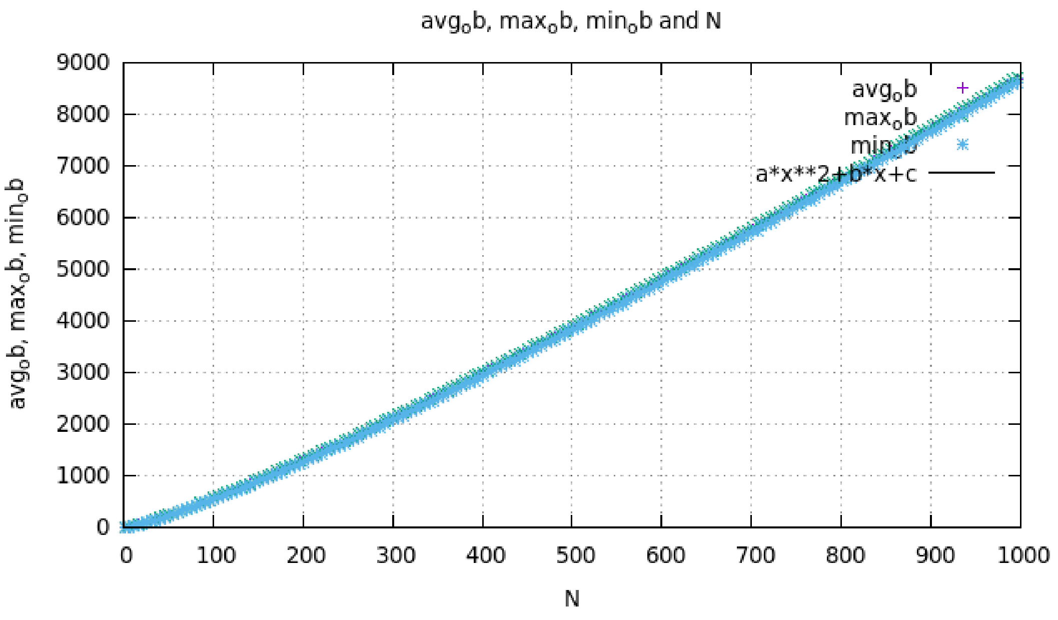
5.2 Section 2

In exercise 2, we were asked to prove the same algorithm as in one but obtaining the table with the average clock time, and the average, minimum and maximum times the BO is executed depending on the size of the permutation. Also I have checked it with valgrind. This is the result using min 1, max 100 and 100 permutations.

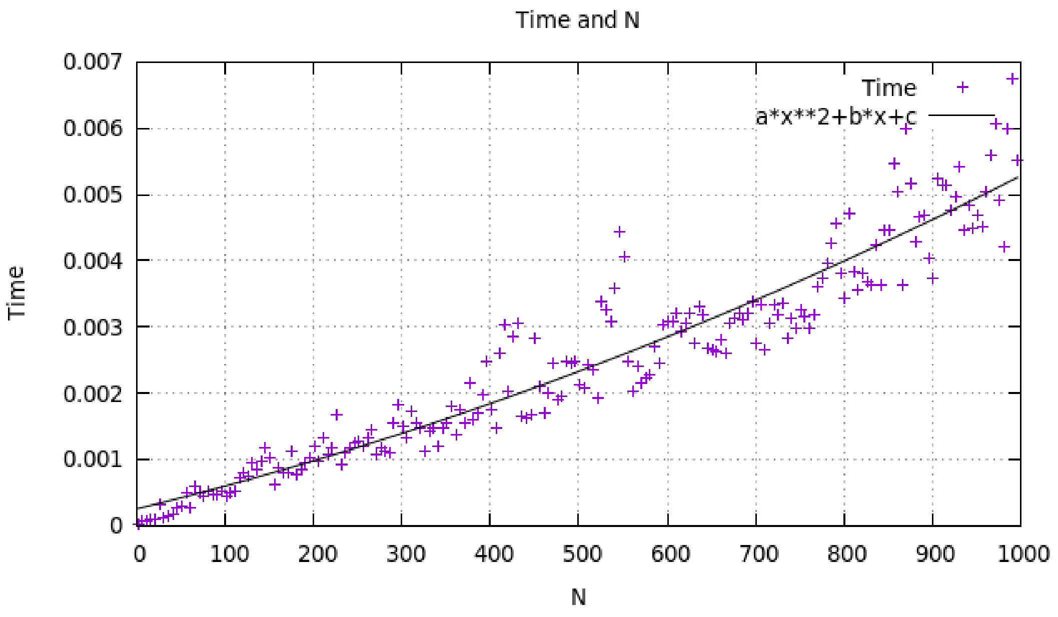


```
xiao@xiao-Lenovo-ideapad-320-15IKB:~/Escritorio/Ing. Inf II/AALG/P2 - FINAL$ valgrind --leak-check=full ./exercise5 -num_min 1 -num_max 100 -i
ncr 5 -numP 100 -outputFile MergeSort
==15701== Memcheck, a memory error detector
==15701== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==15701== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==15701== Command: ./exercise5 -num_min 1 -num_max 100 -incr 5 -numP 100 -outputFile MergeSort
==15701==
Practice number 1, section 5
Done by: your names
Group: Your group
Correct output
==15701==
==15701== HEAP SUMMARY:
==15701==     in use at exit: 0 bytes in 0 blocks
==15701==   total heap usage: 97,024 allocs, 97,024 frees, 2,723,912 bytes allocated
==15701==
==15701== All heap blocks were freed -- no leaks are possible
==15701==
==15701== For counts of detected and suppressed errors, rerun with: -v
==15701== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
xiao@xiao-Lenovo-ideapad-320-15IKB:~/Escritorio/Ing. Inf II/AALG/P2 - FINAL$
```

Plot comparing the best, worst and average number of BOs for MergeSort of size from 0 to 1000. We can observe that the theoretical number and the practical one are almost the same. Also we can see that the best, worst and average number of BOs is similar, but not the same, as MergeSort depends on the size of the table. In the plot it is not so visible but as we have seen in the file generated before, there are differences between the three.

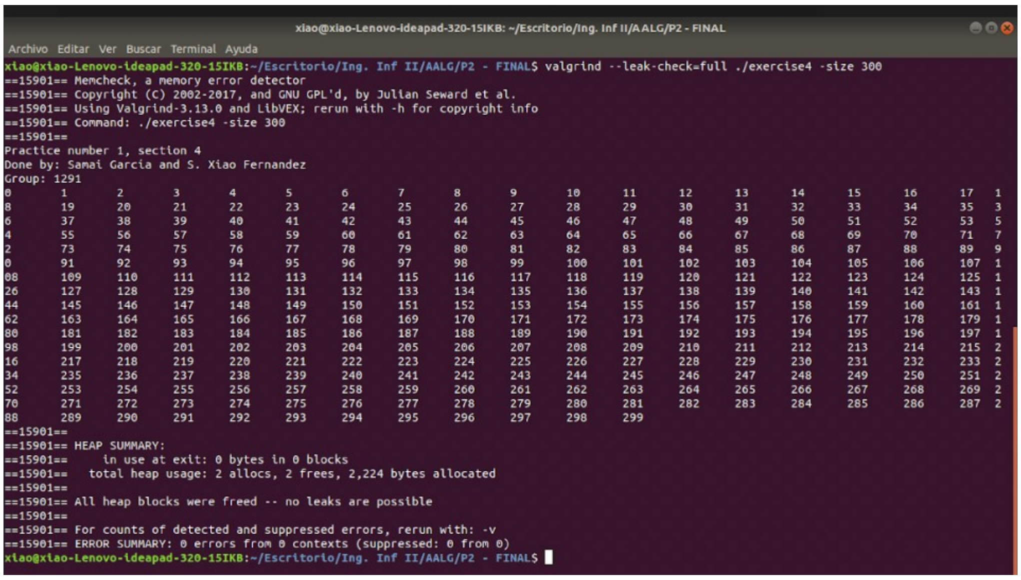


Plot comparing average clock time for MergeSort. In this plot I compare the average clock time with a table of size 1000. We can observe peaks, that is because this algorithm depends a lot in how sorted is the table at the beginning.



5.3 Section 3

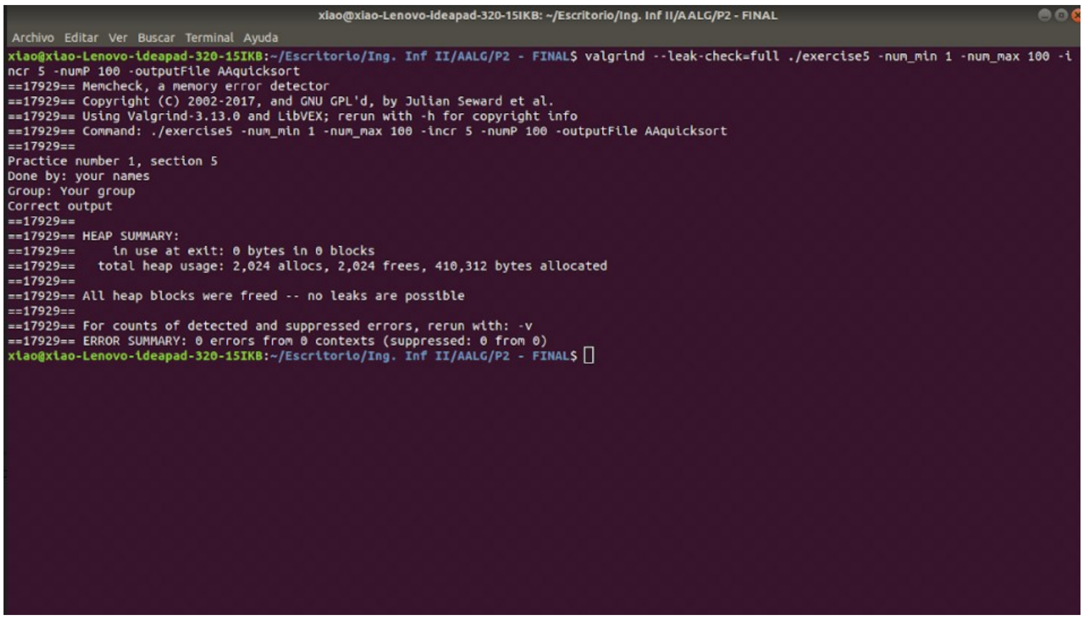
I have checked that the algorithm QuickSort works properly using also file exercise4.c adapted to QuickSort with valgrind checking. This is the result of entering a table of size 300 to be sorted.



```
xiao@xiao-Lenovo-ideapad-320-15IKB: ~/Escritorio/Ing. Inf II/AALG/P2 - FINAL
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
xiao@xiao-Lenovo-ideapad-320-15IKB:~/Escritorio/Ing. Inf II/AALG/P2 - FINAL$ valgrind --leak-check=full ./exercise4 -size 300
==15901== Memcheck, a memory error detector
==15901== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==15901== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==15901== Command: ./exercise4 -size 300
==15901==
Practice number 1, section 4
Done by: Samai Garcia and S. Xiao Fernandez
Group: 1291
0       1       2       3       4       5       6       7       8       9       10      11      12      13      14      15      16      17      1
8       19      20      21      22      23      24      25      26      27      28      29      30      31      32      33      34      35      3
6       37      38      39      40      41      42      43      44      45      46      47      48      49      50      51      52      53      5
4       55      56      57      58      59      60      61      62      63      64      65      66      67      68      69      70      71      7
2       73      74      75      76      77      78      79      80      81      82      83      84      85      86      87      88      89      9
0       91      92      93      94      95      96      97      98      99      100     101     102     103     104     105     106     107     1
08      109     110     111     112     113     114     115     116     117     118     119     120     121     122     123     124     125     1
26      127     128     129     130     131     132     133     134     135     136     137     138     139     140     141     142     143     1
44      145     146     147     148     149     150     151     152     153     154     155     156     157     158     159     160     161     1
62      163     164     165     166     167     168     169     170     171     172     173     174     175     176     177     178     179     1
80      181     182     183     184     185     186     187     188     189     190     191     192     193     194     195     196     197     1
98      199     200     201     202     203     204     205     206     207     208     209     210     211     212     213     214     215     2
16      217     218     219     220     221     222     223     224     225     226     227     228     229     230     231     232     233     2
34      235     236     237     238     239     240     241     242     243     244     245     246     247     248     249     250     251     2
52      253     254     255     256     257     258     259     260     261     262     263     264     265     266     267     268     269     2
70      271     272     273     274     275     276     277     278     279     280     281     282     283     284     285     286     287     2
88      289     290     291     292     293     294     295     296     297     298     299
==15901==
==15901== HEAP SUMMARY:
==15901==     in use at exit: 0 bytes in 0 blocks
==15901==   total heap usage: 2 allocs, 2 frees, 2,224 bytes allocated
==15901==
==15901== All heap blocks were freed -- no leaks are possible
==15901==
==15901== For counts of detected and suppressed errors, rerun with: -v
==15901== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
xiao@xiao-Lenovo-ideapad-320-15IKB:~/Escritorio/Ing. Inf II/AALG/P2 - FINAL$
```

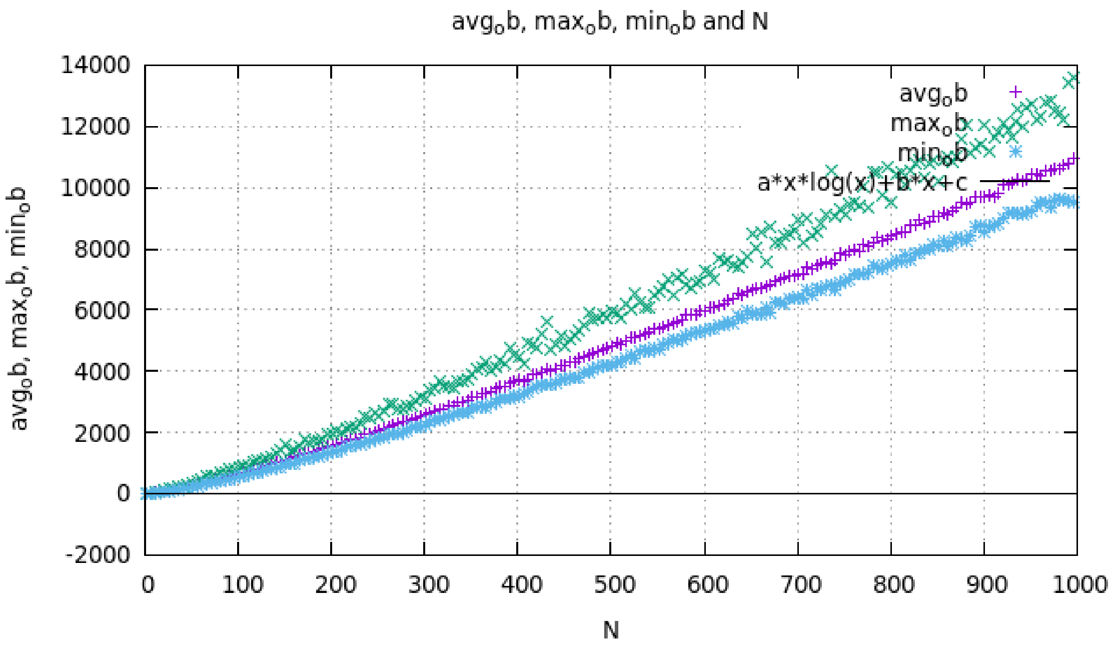
5.4 Section 4

Now I checked algorithm QuickSort with file exercise5.c, which generated a file called "AAquicksort" with the average, minimum and maximum number of times the BO is executed in a table of 100 permutations.

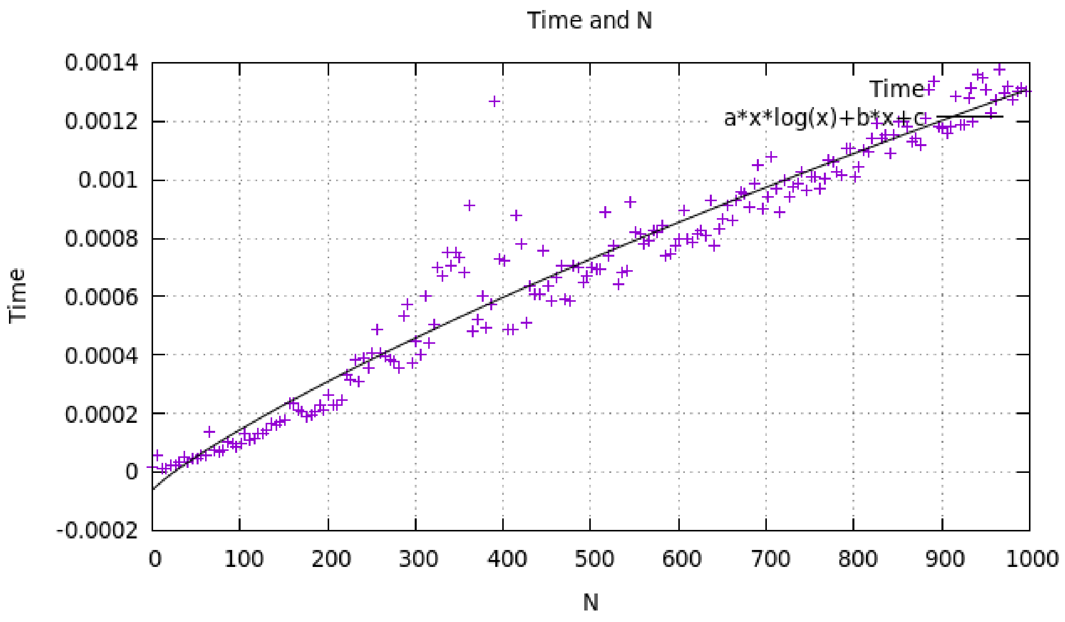


```
xiao@xiao-Lenovo-ideapad-320-15IKB: ~/Escritorio/Ing. Inf II/AALG/P2 - FINAL
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
xiao@xiao-Lenovo-ideapad-320-15IKB:~/Escritorio/Ing. Inf II/AALG/P2 - FINAL$ valgrind --leak-check=full ./exercise5 -num_min 1 -num_max 100 -i
ncr 5 -numP 100 -outputFile AAquicksort
==17929== Memcheck, a memory error detector
==17929== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==17929== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==17929== Command: ./exercise5 -num_min 1 -num_max 100 -incr 5 -numP 100 -outputFile AAquicksort
==17929==
Practice number 1, section 5
Done by: your names
Group: Your group
Correct output
==17929==
==17929== HEAP SUMMARY:
==17929==     in use at exit: 0 bytes in 0 blocks
==17929==   total heap usage: 2,024 allocs, 2,024 frees, 410,312 bytes allocated
==17929==
==17929== All heap blocks were freed -- no leaks are possible
==17929==
==17929== For counts of detected and suppressed errors, rerun with: -v
==17929== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
xiao@xiao-Lenovo-ideapad-320-15IKB:~/Escritorio/Ing. Inf II/AALG/P2 - FINAL$
```

Plot comparing the best, worst and average number of BOs for QuickSort. We can see that three parameters have different results, that is because the algorithm depending on how sorted is the table will choose a particular number, with whom it will do swap, and that number is not always the same.

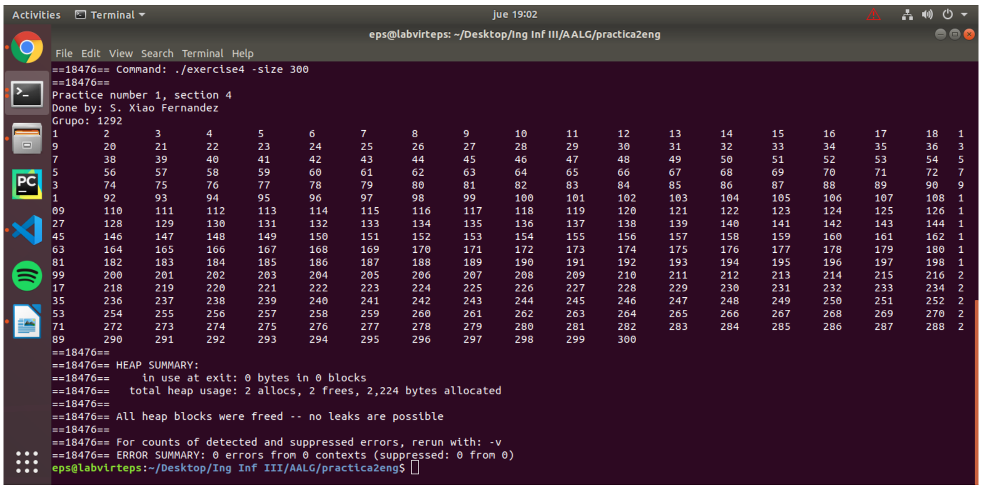


Plot comparing average clock time for QuickSort. In this plot we have used a table also of size 1000. We can observe some peaks also, as in MergeSort, but it adjusts quite a lot to the theoretical average clock time.



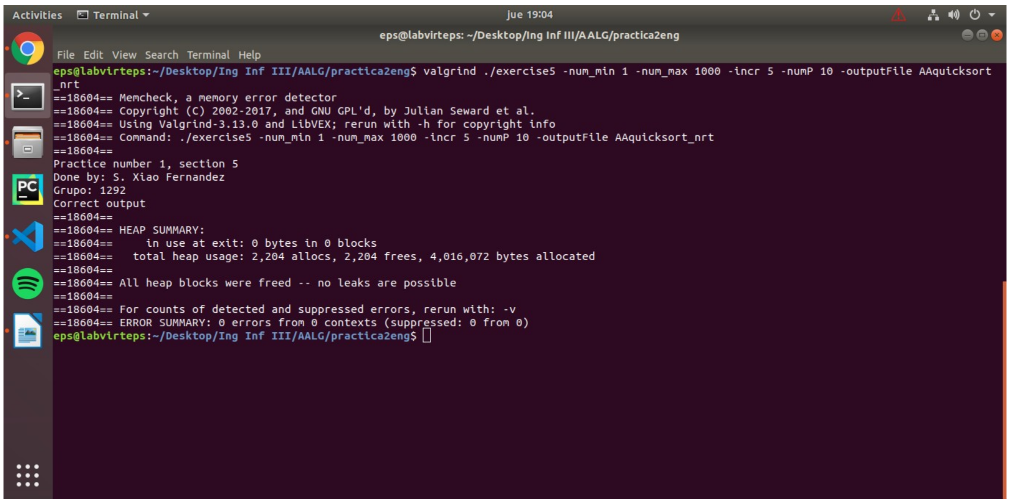
5.5 Section 5

I have checked that the algorithm QuickSort_ntr works properly using also file exercise4.c adapted to QuickSort_ntr with valgrind checking. This is the result of entering a table of size 300 to be sorted.



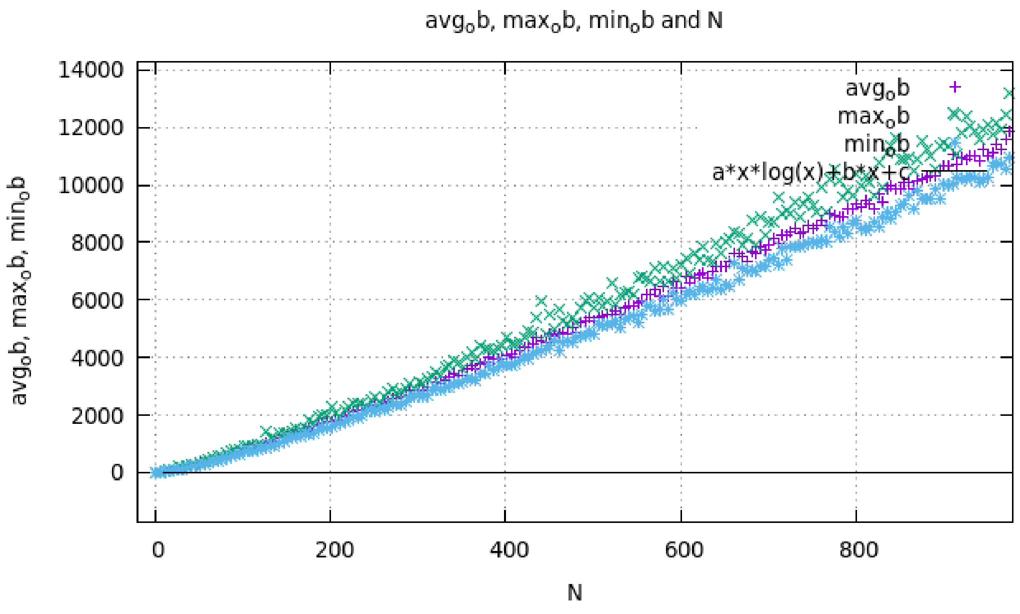
```
==18476== Command: ./exercise4 -size 300
==18476==
Practice number 1, section 4
Done by: S. Xiao Fernandez
Grupo: 1292
1       2       3       4       5       6       7       8       9       10      11      12      13      14      15      16      17      18      1
9       20      21      22      23      24      25      26      27      28      29      30      31      32      33      34      35      36      3
7       38      39      40      41      42      43      44      45      46      47      48      49      50      51      52      53      54      5
5       56      57      58      59      60      61      62      63      64      65      66      67      68      69      70      71      72      7
3       74      75      76      77      78      79      80      81      82      83      84      85      86      87      88      89      90      9
1       92      93      94      95      96      97      98      99      100     101     102     103     104     105     106     107     108     1
09      110     111     112     113     114     115     116     117     118     119     120     121     122     123     124     125     126     1
27      128     129     130     131     132     133     134     135     136     137     138     139     140     141     142     143     144     1
45      146     147     148     149     150     151     152     153     154     155     156     157     158     159     160     161     162     1
63      164     165     166     167     168     169     170     171     172     173     174     175     176     177     178     179     180     1
81      182     183     184     185     186     187     188     189     190     191     192     193     194     195     196     197     198     1
99      200     201     202     203     204     205     206     207     208     209     210     211     212     213     214     215     216     2
17      218     219     220     221     222     223     224     225     226     227     228     229     230     231     232     233     234     2
35      236     237     238     239     240     241     242     243     244     245     246     247     248     249     250     251     252     2
53      254     255     256     257     258     259     260     261     262     263     264     265     266     267     268     269     270     2
71      272     273     274     275     276     277     278     279     280     281     282     283     284     285     286     287     288     2
89      290     291     292     293     294     295     296     297     298     299     300
==18476==
==18476== HEAP SUMMARY:
==18476==     in use at exit: 0 bytes in 0 blocks
==18476==   total heap usage: 2 allocs, 2 frees, 2,224 bytes allocated
==18476==
==18476== All heap blocks were freed -- no leaks are possible
==18476==
==18476== For counts of detected and suppressed errors, rerun with: -v
==18476== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
eps@labvirteps:~/Desktop/Ing Inf III/AALG/practica2eng$
```

Now I checked algorithm QuickSort_ntr with file exercise5.c, which generated a file called "AAquicksort_ntr" with the average, minimum and maximum number of times the BO is executed in a table of 10 permutations.

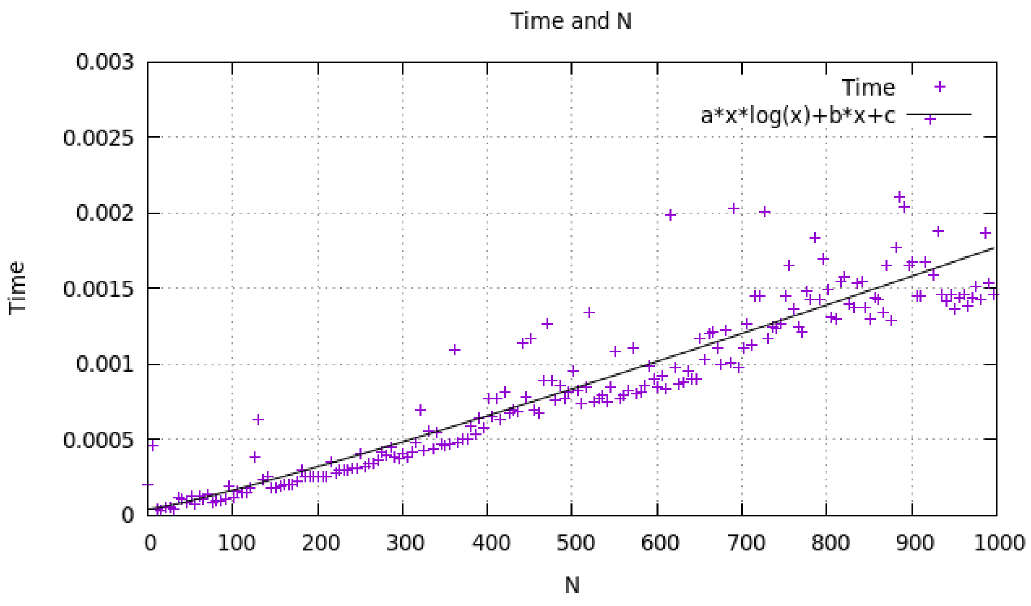


```
eps@labvirteps:~/Desktop/Ing Inf III/AALG/practica2eng$ valgrind ./exercise5 -num_min 1 -num_max 1000 -incr 5 -numP 10 -outputFile AAquicksort
_nrt
==18604== Memcheck, a memory error detector
==18604== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==18604== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==18604== Command: ./exercise5 -num_min 1 -num_max 1000 -incr 5 -numP 10 -outputFile AAquicksort_nrt
==18604==
Practice number 1, section 5
Done by: S. Xiao Fernandez
Grupo: 1292
Correct output
==18604==
==18604== HEAP SUMMARY:
==18604==     in use at exit: 0 bytes in 0 blocks
==18604==   total heap usage: 2,204 allocs, 2,204 frees, 4,016,072 bytes allocated
==18604==
==18604== All heap blocks were freed -- no leaks are possible
==18604==
==18604== For counts of detected and suppressed errors, rerun with: -v
==18604== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
eps@labvirteps:~/Desktop/Ing Inf III/AALG/practica2eng$
```

Plot comparing the best, worst and average number of BOs for QuickSort_ntr. We can see that three parameters have different results, that is because the algorithm depending on how sorted is the table will choose a particular number, with whom it will do swap, and that number is not always the same.



Plot comparing average clock time for QuickSort. In this plot we have used a table also of size 1000. We can observe some peaks also, as in MergeSort, but it adjusts quite a lot to the theoretical average clock time.



5. Answers to theoretical Questions.

5.1 Question 1: **Compare the empirical performance of the algorithms with the theoretical average case for each case. If the traces of the performance graphs are very sharp, why do you think this happens?**

- *MergeSort*: As we have seen in the time plots, the practical time is very similar to the theoretical average case (in the case of MergeSort is N(log(N)). We can also observe that in the plot there are some sharp peaks, this was expected because this algorithm depends a lot in how bad sorted is the table, as it does not go over the complete table. So due to the randomness of the tables we generate, is normal that the graphs are sharp.
- *QuickSort*: On the other hand, QuickSort plots are also sharp, but considerable less than in MergeSort. We can say that in terms of time, MergeSort is quite more efficient than QuickSort, specially when sorting big tables, where QuickSort is less efficient as its average case is 2N log (N) + O(N) .

5.2 Question 2: **Analyze the result obtained when comparing quicksort and quicksort_src both in the case you observe diferences or not.**

As we can see in the plots, the quicksort_ntr time does not have much diferences between the quicksort with the tail recurssion. We see that the quicksort_ntr have bigger peaks of time than the one with the tail recurssion, but does not mean anything because it depends lot of how the initial table is sorted.

When talking about the obs, we can see that the quicksort_ntr is closer to the average ob than the quicksort one.

5.3 Question 3: **What are the best and worst cases for each algorithm? What should be modified in practice to strictly calculate each case (include average case too)?**

MergeSort:

- Best case = ½ * (N log (N))
- Worst case = N log (N) + O (N)
- Average case = N log (N)

QuickSort:

- Best case = We have not seen this in theory class, but knowing how the algorithm works, we can conclude that the best case for QuickSort is when the pivot is in the middle of the table.
- Worst case = (N2/2) - (N/2)
- Average case = 2N log (N) + O (N)

The modifications we should do to strictly calculate each case, are basically found in the functions generate_perms and generate_permutations, because for example to take the table sorted we have to eliminate the loop in charge of doing swap between each of the elements of the table randomly.

5.4 Question 4: **Which of the two algorithms studied is empirically more efficient? Compare this result with the theoretical prediction. Which algorithm(s) is/are more efficient from the point of view of memory management? Justify your answer.**

As seen before, the average case of both algorithms is similar, being MergeSort some better. However, the problem of QuickSort lays in the big size tables, because as we have seen, it is of order N2/2 + O(N), the same as the local sorting algorithms or even MergeSort.

However, if we have to decide between one of them, we will choose QuickSort, at it does not require additional memory, what makes it more efficient and avoids us possible memory errors such as segmentation fault, memory loss or problems allocating.

6. Final Conclusions.

As we have seen in this practice, algorithm MergeSort is the most effective compared to InsertSort and QuickSort, but QuickSort is the most efficient.

On the other hand, it is important for us that the algorithms are the less complex to understand, and QuickSort is not an easy algorithm to understand. Also in QuickSort you have to take into account the position of the pivot because this affects the result.

It has been interesting to create the function quicksort_ntr, that makes the algorithm have less basic operations than the quicksort. This has taught as that making a little modification on the code makes big changes on an algorithm.

Another way could have been modifying the split and median functions.

The first one by returning the average table position as the position of the pivot, and the number of basic operations that it performs is limited to just one.

The second one, choosing the position of the intermediate value between the first element of the table, the last and the one that it is in the intermediate position of the table. Obviously this routine takes a greater number of basic operations. So although some time is lost choosing the pivot, this at the end will result more efficient because having a better pivot will decrease the execution time as the algorithm will sort faster.