Analysis of Algorithms 2020/2021

Practice 1

Pablo Almarza Marques and Miguel Arnaiz Montes

Group 3

Code	Plots	Memory	Total

1. Introduction.

In this practice we will have to implement diverse functions to create random numbers, permutations, sorting algorithms and to calculate the time it takes to execute an algorithm.

2. Objectives.

2.1 Section 1

We will have to work with rand() function and RAND_MAX macro provided by stdlib library to create a function int random_num (int inf, int sup) that generates equiprobable random numbers between the integers inf, sup, both inclusive.

2.2 Section 2

We will have to implement the function int * generate_perm (int N) which will have to implement the following pseudocode:

for i from 1 to N:
    perm[i] = i; for i from 1 to N:
    swap perm[i] with perm[random_num(i, N)];

This will create an array of N numbers randomly generated from 1 to N randomly ordered using section 1's function.

2.3 Section 3

In this third section we will have to create a new function int ** generate_permutations (int n_perms, int N) which will have to create a 2 dimension array storing in each row of the array a permutation of the type of section 2. Having n_perms as the number of permutations stored and N as the number of element inside each of the permutations.

2.4 Section 4

The objective of this section consists on building a new function int InsertSort (int * table, int ip, int iu) which will have to perform the InsertSort algorithm for an array of numbers randomly ordered. Having ip as the first position of the table and iu as the last position of the table.

2.5 Section 5

In this section we will work with the clock functions and pointers in order to come with the time a specified sorting function takes to order a unarranged array of a given length too. It should give the average, minimum and maximum times it has taken.

2.6 Section 6

The objective of this section consists on creating a new function int InsertSortInv (int * table, int ip, int iu) which will have to order the numbers of the table given the other way around we did with section 5 function.

3 Tools and Methodology.

We have used Visual Studio on Linux. Using C/C++ and LiveShare extensions for VS and Valgrind when executing the programs to check errors and memory leaks.

3.1 Section 1

We thought about using rand() and then operating the resulting number to have the random number in the range of numbers we want. For this we operated rand() by dividing by RAND_MAX/(sup-inf+1) and then adding the inferior limit. With this implementation we had the problem that if rand() was equal to RAND_MAX, then the number would be sup+1 not being in the range of numbers we wanted. For this problem we divided rand() by (RAND_MAX+1)/(sup-inf+1) This way the problem is solved.

3.2 Section 2

In this section we did not have any problem. We just traduced the pseudocode into C code without any problem.

3.3 Section 3

In this section we just created a 2 dimension array and allocated memory for it.Then we made use of the previously created functions.

3.4 Section 4

We implemented the InsertSort algorithm getting help from what we have seen in theory class about this algorithm.

3.5 Section 5

For this section we had to learn about the clock functions in order to get the times, asides from using the pointers to a new public structure correctly (we looked for help with both on the internet). Also we had to use calls to previous and new functions which wasn’t really that much of a deal butstill.

3.6 Section 6

We took the function of InsertSort and changing the key comparation of the algorithm, we got to create the inverse of InsertSort.

4. Sourcecode.

4.1 Section 1

```
int random_num(int inf, int sup){
  if (sup < inf) {
    return ERR;
  }
  return (int) ((double)(sup-inf+1) * ((double)rand() / (double)RAND_MAX) + inf);
}
```

4.1 Section 2

```
int* generate_perm(int N) {
  int i, *perm;

  if(N <= 0) return NULL;

  perm = (int*) malloc(N * sizeof(perm[0]));
  if(perm == NULL) return NULL;

  for(i = 0; i < N; i++) perm[i] = i+1;

  for(i = 0; i < N; i++) swap_perm(&perm[i], &perm[random_num(i, N-1)]);
  return perm;
}
```

4.2 Section 3

```
int** generate_permutations(int n_perms, int N) {
  int i, j, *perm, **perms;

  if (n_perms  < 0 || N <= 0) {
    return NULL;
  }
  perms = (int**) malloc(n_perms * sizeof(int*));

 if (perms == NULL) {
    return NULL;
  }
  for(i = 0; i < n_perms; i++) {
    /* Safe way to manage errors to prevent memory leaks */
    perm = generate_perm(N);
    if(!perm){
      /* Optimized to avoid doing N checks*/
      for(j = 0; j < i; j++){
        free(perms[j]);
      }
      free(perms);
      return NULL;
    }
    perms[i] = perm;
  }
  return perms;
}
```

4.3 Section 4

```
int InsertSort(int* table, int ip, int iu){
  int i, j, k, count = 0;

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

  for (i = ip + 1; i <= iu; i++){
    k = table[i];
    j = i-1;

    while (j >= ip && bo(table[j] > k, &count)){
      table[j+1] = table[j];
      j--;
    }
    table[j+1] = k;
  }

  return count;
}
```

4.4 Section 5

```
short average_sorting_time(pfunc_sort method,
                           int n_perms,
                           int N,
                       PTIME_AA ptime)
{
  clock_t begin, end;
  int **perm=NULL, i, j, aux, n_ob=0;
  double time;
  if(!ptime || !method || n_perms<0 || N<=0){return ERR;}

  ptime->average_ob=0;
  ptime->max_ob=0;
  ptime->min_ob=0;
  ptime->time=0;

  perm = generate_permutations(n_perms, N);
  if(!perm){return ERR;}

  begin=clock();
  for(i=0; i<n_perms; i++){
    aux = method(perm[i], 0, N-1);
    if(aux==ERR){
      for(j=0 ; j<i ; j++){
        free(perm[j]);
      }
      free(perm);
      return ERR;
    }
    if(aux<ptime->min_ob || ptime->min_ob == 0){ptime->min_ob = aux;}
    if(aux>ptime->max_ob || ptime->max_ob == 0){ptime->max_ob = aux;}
    n_ob+=aux;
  }
```

```
    end=clock();

    time=(double)(end-begin)/CLOCKS_PER_SEC;
    time=time/n_perms;

    ptime->time=time;
    ptime->average_ob= (double)(n_ob)/n_perms;
    ptime->N=N;
    ptime->n_elems=n_perms;

    for(i=0 ; i<n_perms ;i++){
      free(perm[i]);
    }
    free(perm);

    return OK;
  }


short generate_sorting_times(pfunc_sort method, char* file,
                             int num_min, int num_max,
                             int incr, int n_perms)
{
  PTIME_AA ptime=NULL;
  int i, j, n;

   if(!method || !file || num_min<0 || num_max<num_min || n_perms<0 || incr<=0){return
   ERR;}

  n=((num_max-num_min)/incr)+1;
  ptime = malloc(n*sizeof(TIME_AA));
  if(!ptime){return ERR;}
  for(i=num_min, j=0; i<=num_max ; i+=incr, j++){
   if(average_sorting_time(method, n_perms, i, &ptime[j])==ERR){
     free(ptime);
     return ERR;
   }
  }
  if(save_time_table(file, ptime, n)==ERR){
    free(ptime);
    return ERR;
  }

  free(ptime);

  return OK;
}
```

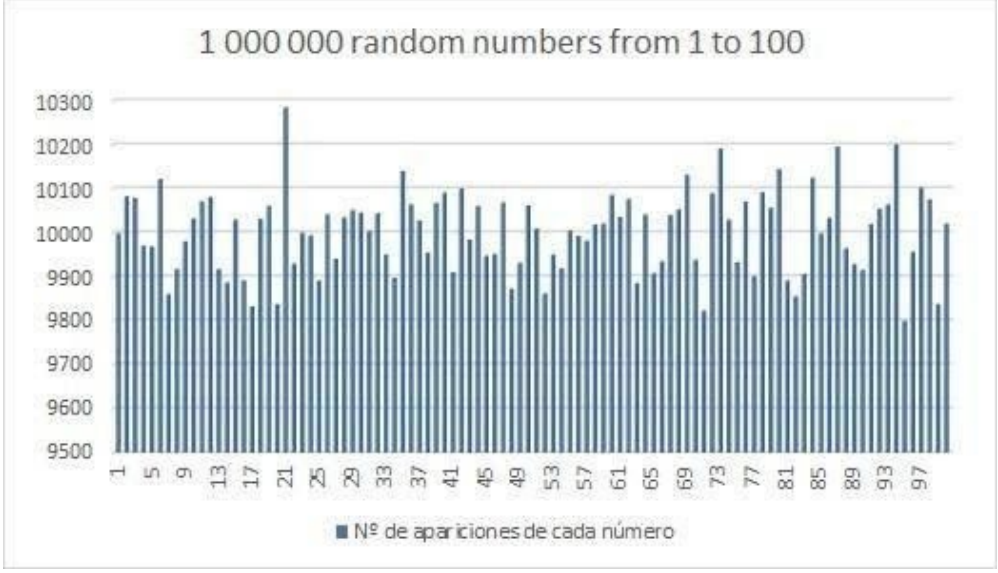
4.5 Section 6

```
int InsertSortInv(int* table, int ip, int iu)
{
  int i, j, k, count = 0;

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

  for (i = ip + 1; i <= iu; i++)
  {
    k = table[i];
    j = i-1;

    while (j >= ip && bo(table[j] < k, &count)){
      table[j+1] = table[j];
      j--;
    }
    table[j+1] = k;
  }

  return count;
}
```

5. Results, Plots.

5.1 Section 1

./exercise1 -limInf 1 -limSup 100 -numN 1000000 | sort -n | uniq -c



5.2 Section 2

5.3 ./exercise2 -size 20 -numP8 Practice number 1, section2

Done by: Pablo Almarza and Miguel Arnaiz

Group: 1292

8 16 11 5 14 0 2 9 18 19 10 7 4 3 13 6 12 15 1 17

10 7 0 12 18 14 1 17 2 15 13 6 9 19 4 3 16 11 5 8

4 7 19 12 0 15 3 16 5 14 18 6 10 8 11 9 2 1 17 13
9 1 17 18 3 15 6 7 4 12 10 0 2 13 8 14 5 11 16 19
13 6 10 12 14 8 2 4 5 19 17 15 9 1 16 3 0 18 11 7
16 18 8 11 19 5 9 15 10 0 4 1 12 6 13 3 7 2 14 17
12 1 7 0 14 16 11 19 6 8 9 10 13 5 17 15 18 2 4 3
10 15 13 12 7 5 9 19 16 14 8 6 0 4 2 1 11 17 18 3

5.4 Section 3

./exercise3 -size 20 -numP8 Practice number 1, section3
Done by: Pablo Almarza and Miguel Arnaiz
Group: 1292
2 0 12 19 3 5 11 13 16 9 10 18 15 8 4 7 6 1 17 14
17 12 11 1 6 13 0 15 7 18 2 4 19 9 5 3 8 10 14 16
3 8 1 12 17 2 18 0 6 9 13 4 16 5 15 10 11 7 14 19
19 11 16 18 8 12 17 4 15 1 2 3 10 0 5 6 14 7 13 9
5 4 16 18 1 6 14 13 11 19 15 7 2 10 0 9 8 12 17 3
2 10 7 15 11 3 8 5 19 9 17 13 4 1 0 16 18 6 12 14
16 5 1 8 12 10 3 11 7 9 18 14 0 13 6 15 19 17 4 2
7 10 4 13 5 9 18 17 2 15 0 19 6 11 8 16 1 12 3 14

5.5 Section 4

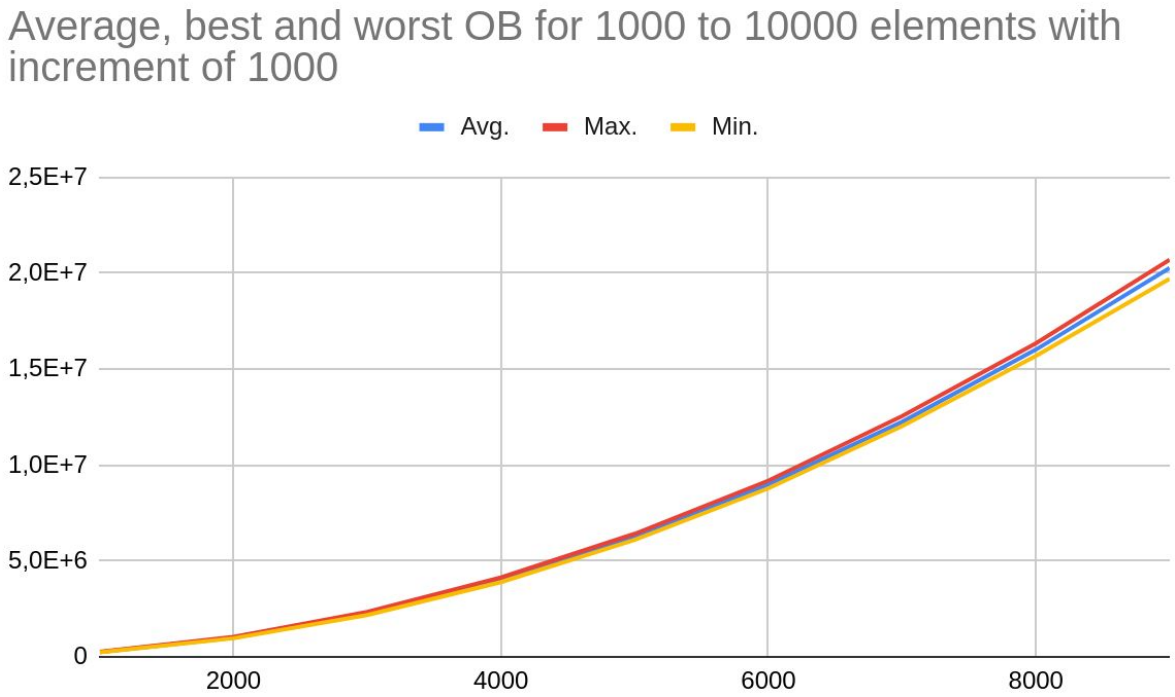
./exercise4 -size20

Practice number 1, section 4
Done by: Pablo Almarza and Miguel Arnaiz
Group: 1292
0 1 2 3 4 5 6 7 8 9 10 11
12 13 14 15 16 17 18 19

5.6 Section

valgrind ./exercise5 -num_min 1000 -num_max 10000 -incr 1000 -numP 100 –outputFile File Practice number 1, section 5
Done by: Pablo Almarza and Miguel Arnaiz
Group: 1292
Correct output

Size	Avg.	Max.	Min.
1000	250632.4	261156	238577
2000	1000536	1036665	966113
3000	2254722	2348001	2180182
4000	4003340	4136827	3895167
5000	6253972	6413271	6099692
6000	8997572	9188200	8785006
7000	12252790	12549709	12026558
8000	16018760	16350321	15687166
9000	20270620	20699274	19708696

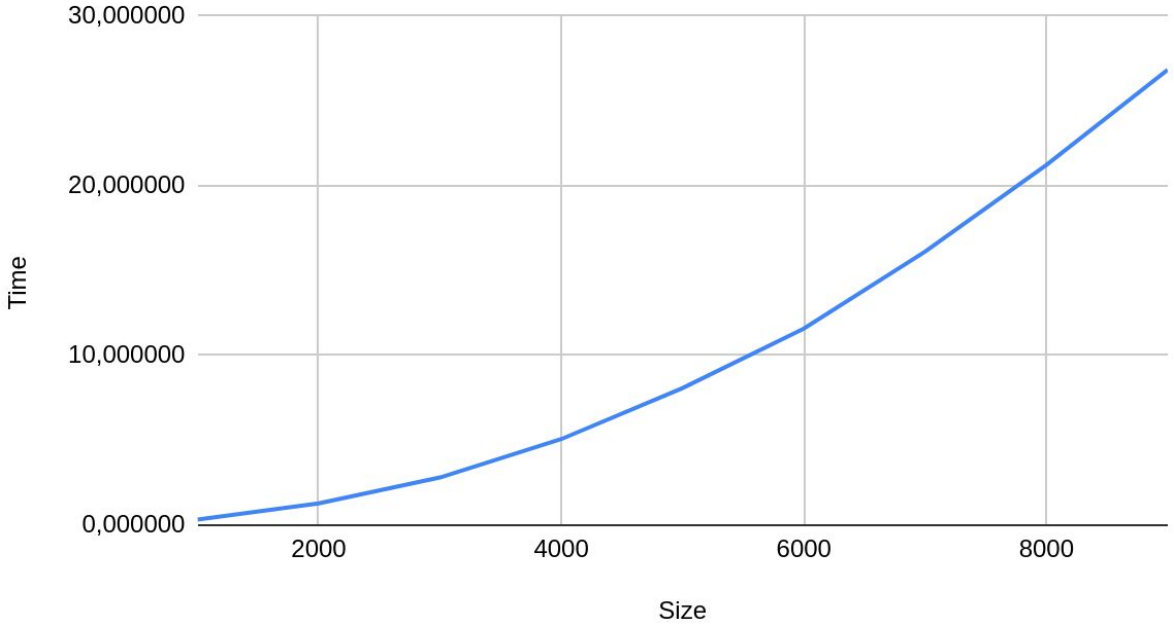


5.7 Section 6

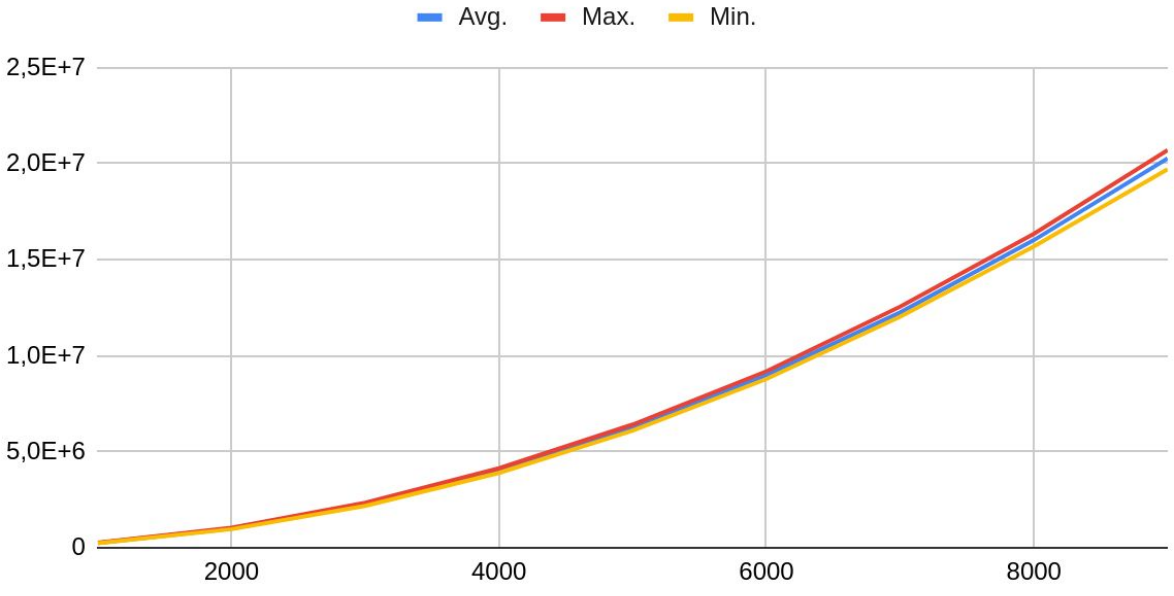
./exercise5 -num_min 25 -num_max 200 -incr 25 -numP 10 -outputFile PFile
Practice number 1, section 5
Done by: Pablo Almarza and Miguel Arnaiz
Group: 1292
Correct output

Size	Time	Avg.	Max.	Min.
1000	0,3266020	2.498313e+05	263007	233447
2000	1,282406	1.002899e+06	1042726	965663
3000	2,809800	2.254707e+06	2311535	2193634
4000	5,069865	4.002627e+06	4113747	3876603
5000	8,076939	6.256168e+06	6407663	6137907
6000	11,58785	9.005702e+06	9187813	8862184
7000	16,12036	1.225248e+07	12463718	12052767
8000	21,22768	1.600871e+07	16261233	15726166
9000	26,83391	2.024405e+07	20609906	19913270

Time vs Size



Average, best and worst OB for 1000 to 10000 elements with increment of 1000



6. Answers to theoretical Questions.

6.1 Question 1

We thought about using rand() and then operating the resulting number to have the random number in the range of numbers we want. For this we operated rand() by dividing by RAND_MAX/(sup-inf+1) and then adding the inferior limit. With this implementation we had the problem that if rand() was equal to RAND_MAX, then the number would be sup+1 not being in the range of numbers we wanted. For this problem we divided rand() by (RAND_MAX+1)/(sup-inf+1) This way the problem is solved.
Another way to do it is using the module operator: num = inf+(rand()%(sup+1))

6.2 Question 2

```
int InsertSort(int* table, int ip, int iu)
{
  int i, j, k, count = 0;

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

  for (i = ip + 1; i <= iu; i++)
  {
    k = table[i];
    j = i-1;

    while (j >= ip && bo(table[j] > k, &count)){
      table[j+1] = table[j];
      j--;
    }
    table[j+1] = k;
  }

  return count;
}
```

Insert Sort compares the index where it is at with the forward elements. If it finds the condition, the number in j goes up. When it reaches ip, the change is produced between the index that was compared with the number in the index of j.

6.3 Question 3

The outer loop of InsertSort does not act on the first element of the table because the idea of the algorithm is comparing an element with its previous element in the array. So taking the first element of the array would have no sense because there is no previous element to the first one.

6.4 Question 4

The basic operation of the InsertSort algorithm is *T[j]>A* being A an element of the array and T[j] the previous element to A.

6.5 Question 5

Because we are studying the InsertSort, the worst case scenario will be when the array is inverted (N, N-1, … , 2, 1) as the algorithm will have to move every single number and the time will be ((N^2)-N)/2. The best case will be when the array is already in order (1,2,...,N-1, N) because it is already ordered so the algorithm will not swap any number and the time will be N-1.

6.5 Question 6

The time it takes for both the InsertSort and the inserSortInv is approximately the same, because the elements are arranged randomly there is no difference in approaching them from the end or the start.

7. Final Conclusions.

To summarize what we have learnt and worked with in this whole project, we have the use of clock functions, the creation of pseudo-random numbers and using them to randomize arrays, as well as the mechanism of the sorting functions and the basic measurement of function times being at its worst best and average cases.