Análisis de Algoritmos 2018/2019

Práctica 1

Andrés Mena Godino y Juan Moreno Díez, Grupo 1272.

Código	Gráficas	Memoria	Total

1. Introducción.

En esta práctica, trataremos de analizar los tiempos de ordenación de los algoritmos SelectSort y SelectSortInv, y compararlos con los datos que hemos estudiado matemáticamente en la parte teórica de la asignatura. El comienzo trata de aprender a generar permutaciones de números aleatorios definidos en un intervalo. Más tarde, nos encargaremos de generar tablas aleatorias, para mas tarde usar los métodos SelectSort y SelectSortInv para ordenar estas tablas. Finalmente compararemos tanto el tiempo medio de OBs como el tiempo medio de reloj de los dos métodos. En algunos casos se harán comprobaciones de los casos mejor y peor también.

2. Objetivos

2.1 Apartado 1

El principal objetivo de este apartado es crear una función llamada: **int aleat_num(int inf, int sup),** la cual genera un un número aleatorio(de tipo entero) equiprobable entre los dos argumentos que le entran a la función: inf y sup(de tipo entero también).

2.2 Apartado 2

El objetivo de este apartado es implementar el pseudocódigo proporcionado en el enunciado para implementar una función. Dicha función es: **int *genera_perm(int N)**, la cual devuelve un puntero a un entero y como argumento le entra un entero que define los elementos de la permutación. En este ejercicio utilizaremos la rutina creada en el apartado anterior, el cual generará los números aleatorios para los elementos.

2.3 Apartado 3

En esta sección implementaremos una función definida como: **int **genera_permutaciones(int n_perms, int N)**. Como argumentos recibe dos enteros, **n_perms**, el cual define el número de permutaciones que queramos que se muestren y **N,** que define el número de elementos que haya en cada permutación. Utilizaremos para ello la rutina creada en el anterior ejercicio **genera_perm**.

2.4 Apartado 4

El desarrollo de este apartado consiste en crear una función llamada: **int SelectSort(int *tabla, int ip, int iu).** Esta función devolverá ERR en caso de de error o el número de veces que se ha ejecutado la OB(operación básica) en el caso de que la tabla se haya

ordenado correctamente. La función recibe tres argumentos, **tabla;** que es la tabla a ordenar, **ip;** que es el primer elemento de la tabla e **iu;** último elemento de la tabla. El algoritmo que queremos implementar utiliza una función auxiliar para encontrar el valor mínimo de subtablas. Por lo tanto es conveniente el desarrollo del prototipo **int minimo(int *tabla, int ip, int iu).**

2.5 Apartado 5

Este apartado consiste en crear tres funciones diferente que están relacionadas entre si: **short tiempo_medio_ordenacion(pfunc_ordena metodo, int n_perms, int N, PTIEMPO ptiempo), short genera_tiempos_ordenacion(pfunc_ordena metodo, char* fichero, int num_min, int num_max, int incr, int n_perms)** y **short guarda_tabla_tiempos(char* fichero, PTIEMPO tiempo, int n_tiempos)**.

La primera función tiene como objetivo rellenar los datos de una estructura de tipo tiempo, en el que incluimos información sobre el "método" que se utiliza, y nos dan el tamaño de la tabla a ordenar y el número de permutaciones. Esta información esta formada por el mínimo y máximo numero de OBs que realiza el "método", el tiempo de ejecución medio, y el número de permutaciones y tamaño de las tablas.

La segunda función reserva memoria para un array de tipo tipo TIEMPO, estructuras que iremos llenando de información mediante la llamada a la función anteriormente mencionada. La cantidad de memoria que se reserva, y el número de veces que se ejecuta la llamada a la función primera, depende de los parámetros de entrada (num_min, num_max, incr). Finalmente, esta función realiza una llamada a guarda_tabla_tiempos, función que se encarga de grabar la información de nuestro array de estructuras en un fichero, que utilizaremos más tarde para realizar las gráficas.

2.6 Apartado 6

El trabajo a realizar en este apartado es muy similar al del apartado 4. El objetivo principal es implementar la rutina **int SelectSortInv(int *tabla, int ip, int iu),** la cual posee los mismos argumentos y retorno que SelectSort, solo que queremos que nos ordene la tabla en orden inverso. Para ello, hemos creado una función **int maximo(int *tabla, int ip, int iu),** la cual devuelve el valor máximo. Con esto y unas modificaciones en la rutina **SelectSort**, los elementos estarán ordenados de forma inversa a los de **SelectSort**.

3. Herramientas y metodología

El entorno de desarrollo en el que se ha llevado a cabo la práctica es **Linux**, ya que ofrece una mejor estructuración y ordenación para el trabajo en C. Como editor de texto se ha utilizado el programa **Atom** para escribir el código. La interfaz de este programa es muy visual y ayuda mucho a organizarse entre los archivos fuente y de cabecera.

3.1 Apartado 1

Para este apartado hemos utilizado el editor de texto **Atom.** Para compilar se ha empleado **gcc** y **Valgrind** para resolver los errores de memoria. Con respecto a las gráficas y el histograma, hemos utilizado **gnuplot**.

3.2 Apartado 2

Para este apartado hemos utilizado el editor de texto **Atom.** Para compilar se ha empleado **gcc** y **Valgrind** para resolver los errores de memoria.

3.3 Apartado 3

Para este apartado hemos utilizado el editor de texto **Atom.** Para compilar se ha empleado **gcc** y **Valgrind** para resolver los errores de memoria.

3.4 Apartado 4

Para este apartado hemos utilizado el editor de texto **Atom.** Para compilar se ha empleado **gcc** y **Valgrind** para resolver los errores de memoria.

3.5 Apartado 5

Para este apartado hemos utilizado el editor de texto **Atom.** Para compilar se ha empleado **gcc** y **Valgrind** para resolver los errores de memoria. Con respecto a las gráficas y el histograma, hemos utilizado **gnuplot**.

3.6 Apartado 6

Para este apartado hemos utilizado el editor de texto **Atom.** Para compilar se ha empleado **gcc** y **Valgrind** para resolver los errores de memoria. Con respecto a las gráficas y el histograma, hemos utilizado **gnuplot**.

4. Código fuente

Aquí ponéis el código fuente **exclusivamente de las rutinas que habéis desarrollado vosotros** en cada apartado.

4.1 Apartado 1

```
int aleat_num(int inf, int sup)
{
  assert(inf >= 0);
  assert(sup >= inf);

  return inf + rand() % (sup-inf+1);
}
```

4.2 Apartado 2

```
void swap(int *p, int *q){
  int d;

  d = *p;
  *p = *q;
  *q = d;
}

int* genera_perm(int N)
{
  int i;
  int* perm;

  perm = (int*)malloc(N*sizeof(int));

  if(!perm)
    return NULL;

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

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

4.3 Apartado 3

```
int** genera_permutaciones(int n_perms, int N)
```

```
{
  int **perm;
  int i, j;

  perm = (int**)malloc(n_perms*sizeof(perm[0]));

  if(perm == NULL)
    return NULL;

  for(i=0; i < n_perms; i++){
    perm[i] = genera_perm(N);
      if(perm[i] == NULL){
        for(j=0; j < i; j++)
        free(perm[j]);

        free(perm);
        return NULL;
      }
  }
  return perm;
}
```

4.4 Apartado 4

```
int SelectSort(int* tabla, int ip, int iu)
{
  int i, cont_obs=0;

  for(i=ip; i<iu; i++){
    int min = minimo(tabla, i, iu);
    swap(&tabla[i], &tabla[min]);
    cont_obs+=(iu-i);
  }
  return cont_obs;
}

int minimo(int *tabla, int ip, int iu){
  int min;
  int i;

  min = ip;

  for(i=ip+1; i <= iu; i++){
    if(tabla[i] < tabla[min])
    min = i;
  }

  return min;
}
```

4.5 Apartado 5

```
short tiempo_medio_ordenacion(pfunc_ordena metodo,
                              int n_perms,
                              int N,
                              PTIEMPO ptiempo)
{
  int i;
  double start, end;
```

```
  int **perms = genera_permutaciones(n_perms, N);
  if(perms == NULL)
    return ERR;

  ptiempo->medio_ob = 0;
  ptiempo->max_ob = 0;
  ptiempo->min_ob = INT_MAX;

  start=clock();
  for (i=0; i<n_perms; i++){
    int obt = metodo(perms[i], 0, N-1);

    ptiempo->medio_ob+=obt;
    if (obt>ptiempo->max_ob)
      ptiempo->max_ob = obt;
    if (obt<ptiempo->min_ob)
      ptiempo->min_ob = obt;
  }
  end=clock();
  ptiempo->medio_ob = (double)(ptiempo->medio_ob/n_perms);
  ptiempo->tiempo = (double)((end-start)/n_perms/CLOCKS_PER_SEC);
  ptiempo->N = N;
  ptiempo->n_elems = n_perms;

  return OK;
}

short genera_tiempos_ordenacion(pfunc_ordena metodo, char* fichero,
                                int num_min, int num_max,
                                int incr, int n_perms)
{
        PTIEMPO ptiempo;
        int j, i;
        for (j=num_min, i=0  ; j<=num_max ; j+=incr, i++);

        ptiempo=(PTIEMPO) malloc ((i-1) * sizeof(ptiempo[0]));
        if (ptiempo==NULL)
                return ERR;



        for (j=num_min, i=0  ; j<=num_max ; j+=incr, i++){
                if (tiempo_medio_ordenacion (metodo, n_perms, j,
&ptiempo[i])==ERR){
     free (ptiempo);
     return ERR;
    }
  }

        if (guarda_tabla_tiempos(fichero, ptiempo, i-1)==ERR){
     free (ptiempo);
     return ERR;
    }
        return OK;
}

short guarda_tabla_tiempos(char* fichero, PTIEMPO tiempo, int n_tiempos)
{
        FILE *f;
```

```
        int i;

        f = fopen (fichero, "w");
        if (f==NULL)
                return ERR;

        for (i=0; i<n_tiempos; i++)
         fprintf (f, "%d\t%f\t%f\t%d\t%d\n", tiempo[i].N, tiempo[i].tiempo,
tiempo[i].medio_ob, tiempo[i].max_ob, tiempo[i].min_ob);

  fclose (f);
        return OK;
}
```

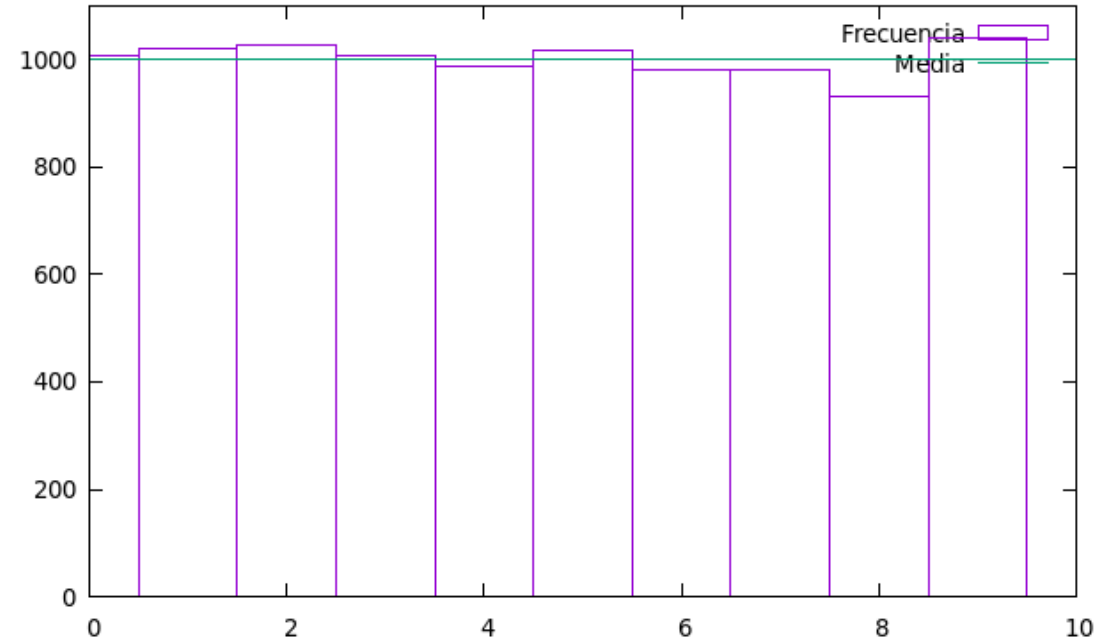
4.6 Apartado 6

```
int SelectSortInv(int* tabla, int ip, int iu)
{
  int i, cont_obs=0;

  for(i=ip; i<iu; i++){
    int max = maximo(tabla, i, iu);
    swap(&tabla[i], &tabla[max]);
    cont_obs+=(iu-i);
  }

  return cont_obs;

}

int maximo(int *tabla, int ip, int iu){
  int max;
  int i;

  max = ip;

  for(i=ip+1; i <= iu; i++){
    if(tabla[i] > tabla[max])
    max = i;
  }

  return max;
}
```

5. Resultados, Gráficas

5.1 Apartado 1

```
mena@Lenovo-Mena:~/Descargas/Andres_Mena_Juan_Moreno$ ./ejercicio1 -limInf 0 -li
mSup 9 -numN 10000 | sort | uniq - c > Ejercicio1.out
mena@Lenovo-Mena:~/Descargas/Andres_Mena_Juan_Moreno$
```



En este historiograma, vemos representado a en el eje de las x el número aleatorio que ha salido (entre el 0 y el 9), y en el eje de las y el número de veces que ha salido cada número. Como podemos observar, casi todos los valores están cerca de la media (se habían realizado 10.000 números aleatorios, entre 10 números distintos, 1000 por cada número). Para que la cantidad de veces que sale cada número fuese aun más equilibrada, habría que generar una mayor cantidad de números aleatorios, por lo que si el número tendiese hacia infinito, todos los números saldrían la misma cantidad de veces

5.2 Apartado 2

```
mena@Lenovo-Mena:~/Descargas/Andres_Mena_Juan_Moreno$ ./ejercicio2 -tamanio 4 -numP 6
Practica numero 1, apartado 2
Realizada por: Andrés Mena y Juan Moreno
Grupo: 9(1272)
4 2 1 3
4 1 2 3
3 2 4 1
2 3 1 4
3 2 4 1
4 2 3 1
mena@Lenovo-Mena:~/Descargas/Andres_Mena_Juan_Moreno$
```

5.3 Apartado 3

```
mena@Lenovo-Mena:~/Descargas/Andres_Mena_Juan_Moreno$ ./ejercicio3 -tamanio 3 -numP 5
Practica numero 1, apartado 3
Realizada por: Andrés Mena y Juan Moreno
Grupo: 9(1272)
1 3 2
3 1 2
2 1 3
3 1 2
3 2 1
mena@Lenovo-Mena:~/Descargas/Andres_Mena_Juan_Moreno$
```

5.4 Apartado 4.1 (SelectSort)

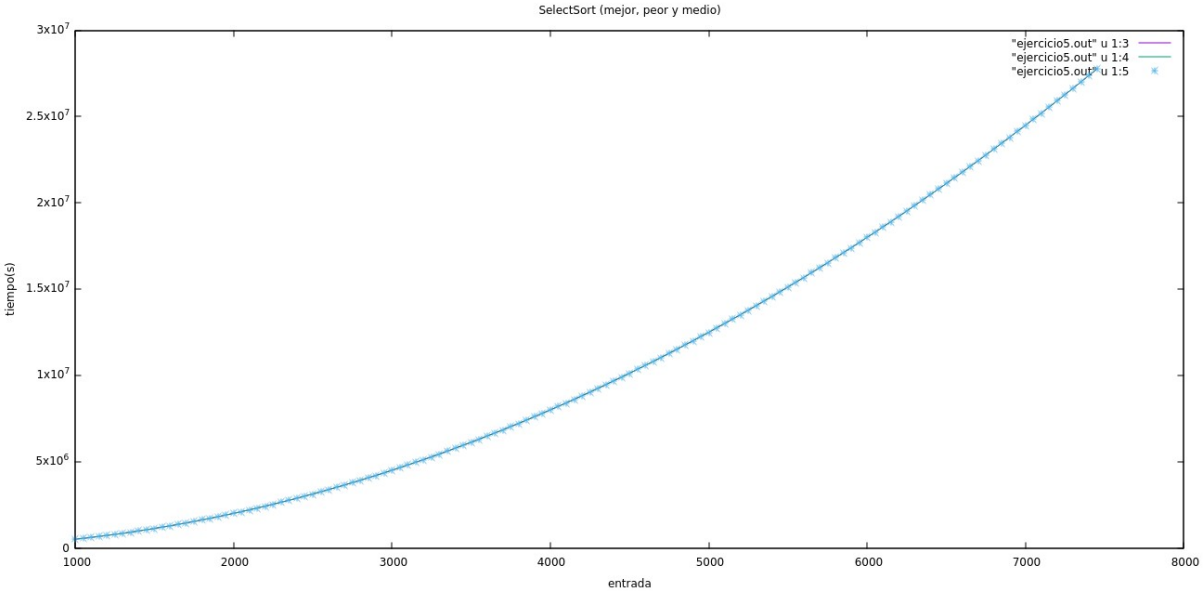
```
mena@Lenovo-Mena:~/Descargas/Andres_Mena_Juan_Moreno$ ./ejercicio4 -tamanio 10
Practica numero 1, apartado 4
Realizada por: Andrés Mena y Juan Moreno
Grupo: 9(1272)
1       2       3       4       5       6       7       8       9       10
mena@Lenovo-Mena:~/Descargas/Andres_Mena_Juan_Moreno$
```

5.4 Apartado 4.2 (SelectSortInv)

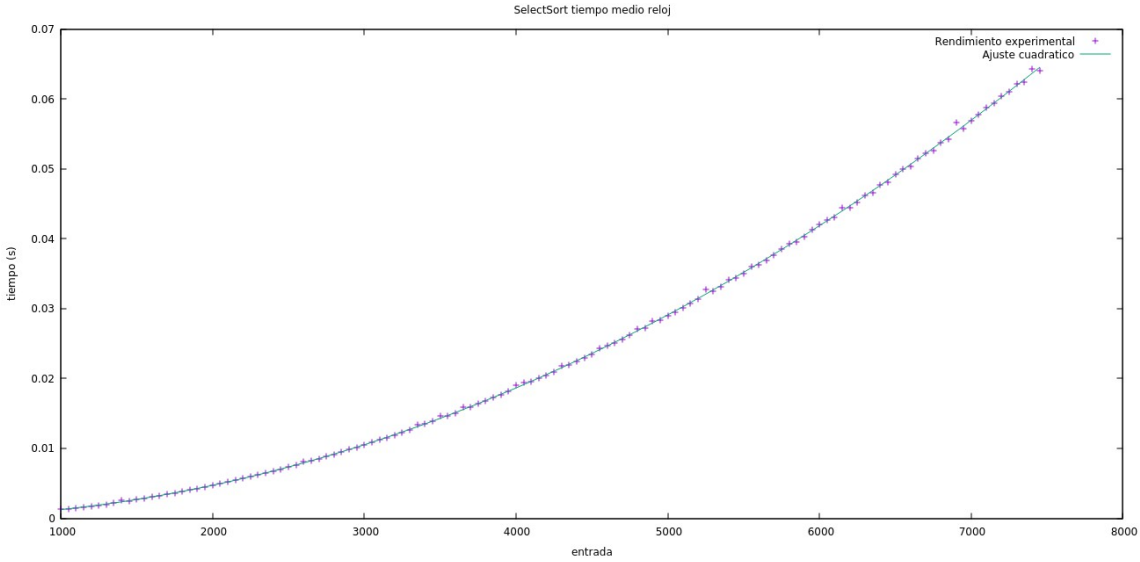
```
mena@Lenovo-Mena:~/Descargas/Andres_Mena_Juan_Moreno$ ./ejercicio6 -tamanio 10
Practica numero 1, apartado 4
Realizada por: Andrés Mena y Juan Moreno
Grupo: 9(1272)
10      9       8       7       6       5       4       3       2       1
mena@Lenovo-Mena:~/Descargas/Andres_Mena_Juan_Moreno$
```

5.5 Apartado 5

```
mena@Lenovo-Mena:~/Descargas/Andres_Mena_Juan_Moreno$ ./ejercicio5 -num_min 1000 -num_max 7500 -incr 50 -numP 50 -fichSalida ejercicio5.out
Practica numero 1, apartado 5
Realizada por: Andrés Mena y Juan Moreno
Grupo: 9(1272)
Salida correcta
mena@Lenovo-Mena:~/Descargas/Andres_Mena_Juan_Moreno$
```



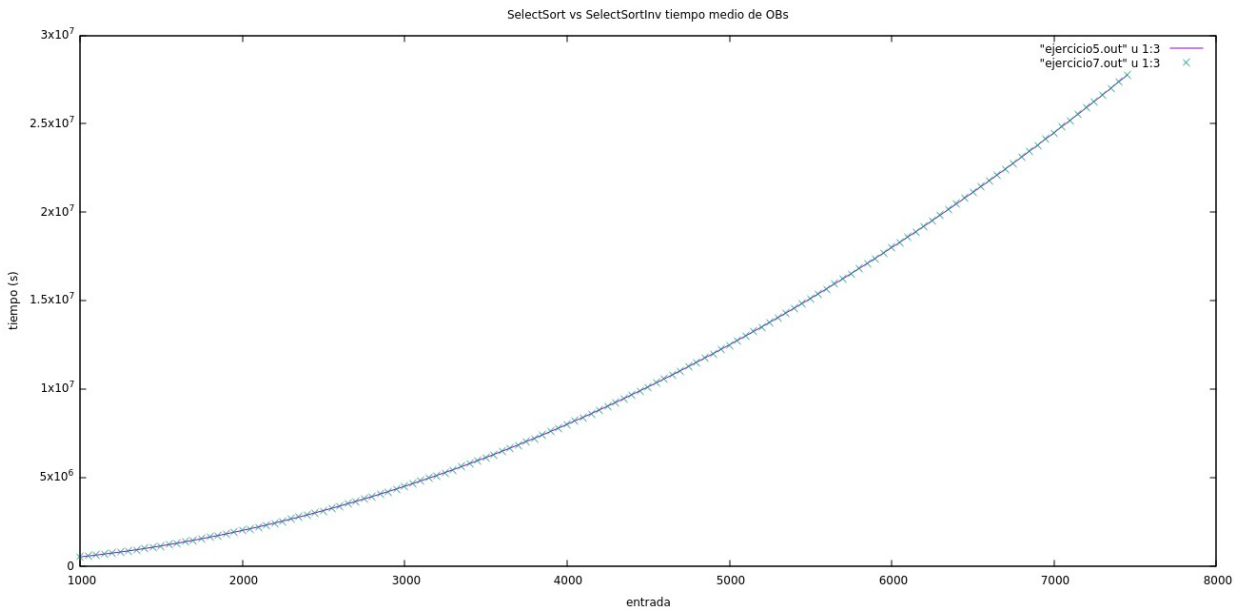
En esta grafica, vemos representados el número de operaciones básicas que se realizan con el algoritmo de SelectSort . Como podemos observar, las tres lineas están superpuestas, es decir, el rendimiento mejor, peor y medio de este algoritmo es el mismo, ya que independientemente de si nos dan una tabla ordenada o completamente desordenada, el número de comparaciones de clave será el mismo, ya que el algoritmo no tiene ningún tipo de flag ni sistema para saber si la tabla ya está ordenada.



Como podemos ver en esta gráfica, se representa el tiempo medio que toma dependiendo de la entrada. Como podemos ver, se ha podido ajustar una función de segundo grado casi perfectamente a los datos experimentales del algoritmo SelectSort. Solo algunos datos se salen de esta linea, y es probablemente como consecuencia de un algún proceso que estuviese realizando el ordenador en segundo plano, lo que hizo que el "cronometro" no fuese tan preciso.

5.6 Apartado 6

```
Salida correcta
mena@Lenovo-Mena:~/Descargas/Andres_Mena_Juan_Moreno$ ./ejercicio7 -num_min 1000 -num_max 7500 -incr 50 -numP 50 -fichSalida ejercicio7.out
Practica numero 1, apartado 7
Realizada por: Andrés Mena y Juan Moreno
Grupo: 9(1272)
Salida correcta
mena@Lenovo-Mena:~/Descargas/Andres_Mena_Juan_Moreno$
```



Vemos en estas gráficas que el tiempo medio de operaciones básicas y el tiempo medio de reloj para el SelectSort y el SelectSortInv son iguales, ya que los dos algoritmos funcionan igual, pero el SelectSort ordena la tabla de menor a mayor, y el SelectSortInv al revés. Esto hace que la única diferencia sea buscar elementos mínimos en vez de máximos, por lo que el rendimiento del algoritmo es idéntico.

5. Respuesta a las preguntas teóricas.

5.1 Para generar la función **aleat_num,** hemos utilizado una página web la cual nos ha ayudado en su implementación(http://www.chuidiang.org/clinux/funciones/rand.php). La idea se basa en generar un número aleatorio entre dos valores máximo y mínimo empleando la siguiente fórmula: numero = M + rand() % (N-M+1), **M** será el valor mínimo y **N** el máximo. En esta formula matemática nos apoyamos en la idea de módulo, la cual nos permite que los valores que se introduzcan en la variable número estén en el rango deseado. Otra implementación podría ser creando una única variable que vaya devolviendo números aleatorios únicamente mediante el uso de rand(); ejemplo: printf("%d", rand());, la desventaja de esta implementación sería que no tendríamos limites inferiores ni superiores sobre los que generar los números aleatorios, y por lo tanto no sería nada precisa.

5.2 El SelectSort ordena bien porque utiliza el siguiente mecanismo: Busca el mínimo de la tabla, y lo intercambia por el primer elemento. A continuación, busca el mínimo otra vez, sin incluir esta vez lo que tenemos en el elemento 1 de la tabla, y lo intercambia con el segundo elemento. Esto se realiza sucesivamente, hasta que la tabla queda completamente ordenada. Es por esto que el algoritmo ordena bien. Ademas, lo hemos comprobado con diferentes entradas, y el resultado que devuelve es una función ordenada.

5.3 El bucle exterior de SelectSort no incluye el último elemento de la tabla porque al llegar al penúltimo elemento, la tabla va a estar ya ordenada y no hace falta realizar más comprobaciones.

5.4 La OB de SelectSort es la comparación de claves: if(tabla[i] < tabla[min]), la cual se encuentra en la función: **minimo**.

5.5 Como hemos demostrado en la parte de teoría, y confirmado de forma práctica con los programas que hemos creado, el rendimiento del algortimo SelectSort (tanto para el caso peor como el caso peor) es $N^2/2 – N/2 \sim N^2/2 + O(N)$. Por lo tanto dependiendo de la entrada, el tiempo de la ejecución ira creciendo de forma cuadrática. Para obtener el tiempo en segundos, miramos la gráfica en la que hemos representado los tiempos medios de reloj.

N=1000	T=0.001340 s

N=2000	T=0.004705 s
N=3000	T=0.010500 s
N=4000	T=0.018989 s
N=5000	T=0.028965 s
N=6000	T=0.041990 s

5.5 Como podemos observar y hemos explicados en los ejercicios previos, las gráficas obtenidas son iguales, y esto es porque los dos algoritmos funcionan de la misma forma, su única diferencia es que el SelectSort busca elementos mínimos de la tabla, y el SelectSortInv busca elementos máximos de la tabla. Es por ello que el números de operaciones básicas va a ser el mismo en ambos casos (tanto en el mejor, como en el peor, como en el medio), y el tiempo de ejecución también será igual.

6. Conclusiones finales.

A lo largo de esta práctica hemos aprendido a como generar números aleatorios en un determinado rango, y luego usando esa función generar un tabla de números aleatorios, para más tarde implementar métodos de ordenación de tablas (SelectSort y SelectSortInv). Ademas hemos comprobado que tal y como habíamos estudiado en teoría se trata de unos algoritmos que tienen un rendimiento cuadrático. El SelectSort, se considera como un algoritmo local, aunque solo lo es moralmente ya que no esta comparando elementos adyacentes tan claro como otros algoritmos locales. Sin embargo, su rendimiento coincide con el rendimiento mínimo de cualquier algoritmo local ($N^2/2 + O(N)$). Esto lo hemos podido comprobar mediante el uso de gráficas, en las que observamos que los valores experimentales que obtenemos mediante el uso de un reloj coinciden con los valores teóricos.