Análisis de Algoritmos 2022/2023

Práctica 1

Javier Jiménez García y Pablo Fernández Izquierdo

Grupo 1201

Código	Gráficas	Memoria	Total

Análisis de Algoritmos 2022/2023

1. Introducción

En esta primera práctica sentamos las bases para analizar algoritmos de ordenación, diseñando funciones básicas de generación de permutaciones aleatorias, ordenación simple y medición de tiempos abstractos/puros ejecución (contando operaciones básicas/tiempo de reloj).

2. Objetivos

2.1 Generación de un número aleatorio (función `random_num()`)

Diseñar e implementar una función que genere un número aleatorio equiprobable en un rango que se toma como argumento, valorando qué estrategia es la más adecuada para garantizar la equiprobabilidad.

2.2 Generación de una permutación aleatoria (función `generate_perm()`)

Implementar una función que genere una permutación aleatoria equiprobable de un número N de elementos (en este caso números naturales de 1 a N) que se toma como argumento. Se aprovecha la función `random_num()` del apartado 1.

2.3 Generación de permutaciones aleatorias (función `generate_perms()`)

Diseñar e implementar una función que genere un número M de permutaciones aleatorias equiprobables de tamaño N, siendo M y N argumentos. Se aprovecha la función `generate_perm()` del apartado 2.

2.4 Ordenación por selección de permutaciones (función `SelectSort()`)

Implementar una función que ordene permutaciones de elementos mediante el método de ordenación por selección, en orden creciente, contando el número de operaciones básicas (CDCs) realizadas en el proceso.

2.5 Medición de tiempos de ejecución (funciones de `times.h`)

Diseñar e implementar funciones que permitan estudiar los tiempos de ejecución de diversos algoritmos de ordenación de permutaciones (en esta práctica $SelectSort$), tanto el abstracto (contando operaciones básicas máximas, mínimas y medias) como el real (midiendo el tiempo que tarda el procesador en ejecutarlo). El proceso consiste en medir experimentalmente estos tiempos, calcular el promedio y exportar estos datos para poder hacer una gráfica que ilustre el coste del algoritmo en función del tamaño de entrada.

2.6 Ordenación por selección inversa (función `SelectSortInv()`)

Implementar en una función el algoritmo de ordenación por selección en orden decreciente, y comparar su eficiencia frente al de orden creciente.

3. Herramientas y metodología

Para el desarrollo de las prácticas estamos utilizando el IDE Visual Studio Code en dos ordenadores portátiles distintos:

1. Un MSI con procesador x86_64 y Windows 11 instalado, compilando y ejecutando los programas en el WSL2 (Ubuntu).
2. Un Macbook con procesador ARM y macOS 12 instalado, compilando y ejecutando los programas en la Terminal de macOS (Unix).

En las sesiones de laboratorio nos conectamos mediante la función Live Share de VS Code para colaborar en tiempo real, y para trabajar por separado tenemos el código en un repositorio de GitHub: Análisis de Algoritmos.

Compilamos con gcc adhiriéndonos al estándar ANSI como indica el enunciado, y comprobamos que las funciones no producen errores ni fugas de memoria ejecutándolas con `valgrind`.

Para las gráficas de resultados hemos utilizado el programa `gnuplot` y scripts en formato `.gp` para procesar los datos obtenidos con programas específicos que los escriben en ficheros.

Finalmente, antes de la entrega, hemos probado que todo se ejecuta correctamente en el entorno de desarrollo de los laboratorios.

3.0 Gráficas de funciones usando Gnuplot

Para agilizar el proceso de crear funciones hemos hecho uso de scripts de Gnuplot. Se trata de ficheros de texto con extensión `.gp`, como el siguiente, que contiene las instrucciones para que Gnuplot genere una gráfica de comparación entre los tiempos de ejecución de $SelectSort$ y $SelectSortInv$:

```
set terminal png size 1000,1000
set output "stats5_comp_1000_time.png"
set title 'Tiempo de ejecución de SelectSort vs SelectSortInv'

set xlabel "Tamaño"
set ylabel "Tiempo (microsegundos)"

plot "stats5_selectsort_1000.txt" using 1:2 title "Tiempo SelectSort" with l lw 2 linecolor 2,\
     "stats5_selectsortinv_1000.txt" using 1:2 title "Tiempo SelectSortInv" with l lw 2 linecolor 1
```

para ejecutarlo se utiliza `$ gnuplot stats5_comp_time.gp`.

También se puede hacer uso de argumentos. En el siguiente caso, ARG1 es el fichero de texto de entrada, ARG2 es el nombre de la imagen de salida y ARG3 es la columna a graficar.

```
set terminal png size 1000,1000
set output ARG2
set title 'Frecuencia de aparición en random\_num'

set xlabel "Número"
set yrange [000000: 24000000]
set ylabel "Frecuencia"

plot ARG1 using 1:int(ARG3) title "Time" with points, 20000000 title "Media esperada" lw 2
```

Para ejecutarlo se utiliza `$ gnuplot -c stats.gp stats5.txt stats5.png 2`.

Incluimos todas estas herramientas en la entrega, sin detrimento de la funcionalidad principal de los materiales que se nos entregaron.

3.1 Generación de un número aleatorio (función `random_num()`)

Para obtener los números en el rango especificado, primero hemos generado un número aleatorio en el intervalo $[0, 1)$ utilizando la función `rand()` y dividiendo el resultado entre `RAND_MAX + 1` (en decimal de precisión doble para no tener problemas aunque `RAND_MAX` sea muy grande, como en nuestro caso que `RAND_MAX = INT_MAX`). Al multiplicar este número por la longitud del intervalo y sumárselo al límite inferior obtenemos un número aleatorio con la misma distribución de probabilidad que las salidas de la función `rand()`.

En el apartado 6.1 discutimos otras posibles aproximaciones a este problema, y justificamos por qué esta es la más adecuada entre ellas.

> ⚠️ **Control de parámetros**:
>
> - Rechazamos los `inf < 0` porque la utilidad de esta función para la práctica es que genere números positivos.
> - Rechazamos los `inf > sup` porque no tiene sentido que el límite inferior esté por encima del superior.

3.2 Generación de una permutación aleatoria (función `generate_perm()`)

Basándonos en el pseudocódigo proporcionado en el enunciado de la práctica, hemos implementado una función generadora de permutaciones. Dicha función acepta como parámetro de entrada el tamaño de la permutación y devuelve

un array de enteros con la permutación resultante.

> ⚠️ **Control de parámetros**:
>
> - Rechazamos los `N <= 0` porque una permutación de ningún elemento o de un número negativo de elementos no tiene sentido.

Pseudocódigo

```
for i from 1 to N:
  perm[i] = i

for i from 1 to N:
  swap(perm[i], perm[random_num(i, N)]
```

3.3 Generación de permutaciones aleatorias (función `generate_perms()`)

La función es bastante simple, reservamos memoria para `n_perms` punteros a entero y en cada uno guardamos la dirección de una permutación del tamaño indicado, generada utilizando la función anterior (`generate_perm()`).

> ⚠️ **Control de parámetros**:
>
> - Rechazamos los `n_perms <= 0` porque generar un número negativo de permutaciones no tiene sentido, y generar cero permutaciones es no hacer nada.
> - Rechazamos los `N <= 0` porque una permutación de ningún elemento o de un número negativo de elementos no tiene sentido.

3.4 Ordenación por selección de permutaciones (función `SelectSort()`)

Hemos implementado el pseudocódigo para el algoritmo $SelectSort$ estudiado en clase de teoría, para ordenar permutaciones en orden ascendente. En el apartado 6.2, explicamos el funcionamiento de este algoritmo y por qué es eficaz (que no eficiente) en la ordenación.

Para el cálculo de las operaciones básicas (CDCs), hemos evitado modificar la función `min()` proporcionada haciendo el "cálculo teórico", aprovechando que $SelectSort$ siempre realiza el mismo número de OBs independientemente del grado de desorden de la entrada. Para cada iteración i, el algoritmo busca el mínimo comparando el elemento i con todos los que quedan a su derecha, es decir, con $U - i$ elementos. Calculamos el total sumando estos valores en cada iteración.

> ⚠️ **Control de parámetros**:
>
> - Rechazamos los `array == NULL` porque no hay tabla de elementos que ordenar.
> - Rechazamos los `ip < 0` porque un índice negativo para una tabla no tiene sentido (al menos en C).

Pseudocódigo

```
SelectSort(array T, P, U):
  for i from P to U - 1:    # i increasing
    min = min(T, i, U)
    swap(T[i], T[min])
```

3.5 Medición de tiempos de ejecución (funciones de `times.h`)

Implementamos las funciones necesarias para analizar el rendimiento de una determinada función de ordenación y exportar los resultados para su posterior análisis.

Función `average_sorting_time()`

Esta función se encarga de hacer tests a una función de ordenación, midiendo el tiempo medio de ejecución y las operaciones básicas máximas, mínimas y la media. Los tests se realizan con `n_perms` permutaciones aleatorias de `N` elementos.

Para medir el tiempo hemos decidido utilizar la función `clock()` de la librería `time.h` cuya unidad de medida son los ticks de reloj. En nuestro caso esto se traduce en microsegundos, pero puede variar dependiendo de la máquina, ya que es la constante `CLOCKS_PER_SECOND` la que marca dicha unidad (en nuestro entorno de desarrollo y en el de los laboratorios tiene un valor de 1.000.000). Para medir los tiempos hemos llamado a la función al principio y al final de los test, es decir, obtenemos el tiempo total y luego dividimos entre el número de test realizados (`n_perms`) para obtener el tiempo medio. Podríamos haber ido midiendo el tiempo que tarda cada test, pero la ejecución es tan rápida que en permutaciones pequeñas no llega a pasar un tick de reloj.

Por último, los datos se guardan en una estructura del tipo `TIME_AA`.

> ⚠ **Control de parámetros**:
>
> - Rechazamos los `metodo == NULL` porque no hay función de ordenación.
> - Rechazamos los `n_perms <= 0` porque generar un número negativo de permutaciones no tiene sentido, y generar cero permutaciones es no hacer nada.
> - Rechazamos los `N <= 0` porque una permutación de ningún elemento o de un número negativo de elementos no tiene sentido.
> - Rechazamos los `ptime == NULL` porque no hay estructura donde guardar los datos del test.

Función `generate_sorting_times()`

Esta función se encarga de hacer tests a una función de ordenación, probando permutaciones de distintos tamaños para poder ver su complejidad y exportar los datos.

Se reserva un array de estructuras `TIME_AA` de tamaño `(num_max - num_min) / incr + 1`, el número de tamaños de permutación a probar, donde se guardarán los datos generados por la función `average_sorting_time()`.

Se realizan test con permutaciones desde `num_min` elementos hasta `num_max` con incrementos de `incr` utilizando la función descrita anteriormente.

Finalmente los datos se guardan en un fichero de texto utilizando la función `save_time_table()` para su posterior análisis por parte del usuario.

> ⚠ **Control de parámetros**:
>
> - Rechazamos los `method == NULL` porque no hay función de ordenación.
> - Rechazamos los `file == NULL` porque no hay fichero donde guardar la información.
> - Rechazamos los `num_min <= 0` porque ordenar una permutación de un número negativo de elementos o de ningún elemento, no tiene sentido.
> - Rechazamos los `num_max < num_min` porque no tiene sentido que el tamaño máximo sea menor que el tamaño mínimo (estamos trabajando en orden creciente de tamaños).
> - Rechazamos los `incr <= 0` porque queremos probar distintos tamaños (no tiene sentido que el incremento sea 0) y estamos trabajando en orden creciente de tamaños (no tiene sentido que el incremento sea negativo).
> - Rechazamos los `n_perms <= 0` porque generar un número negativo de permutaciones no tiene sentido, y generar cero permutaciones es no hacer nada.

Función `save_time_table()`

Esta función se encarga de guardar en un archivo de texto los datos de un array de estructura `TIME_AA`. La función imprime `n_times` filas con los siguientes datos de la estructura (en ese orden): tamaño de las permutaciones, tiempo medio, promedio de operaciones básicas, su máximo y su mínimo.

> ⚠ **Control de parámetros**:
>
> - Rechazamos los `file == NULL` porque no hay fichero donde guardar la información.
> - Rechazamos los `ptime == NULL` porque no hay información que guardar en el fichero.
> - Rechazamos los `n_times <= 0` porque no tiene sentido guardar la información de un número negativo de elementos ni queremos crear un fichero vacío si no hay ningún elemento.

3.6 Ordenación por selección inversa (función `SelectSortInv()`)

Hemos implementado el algoritmo de $SelectSort$ del apartado 3.4 pero esta vez ordena tuplas de enteros en orden descendente. El algoritmo funciona igual que $SelectSort$ pero ordenando la tabla de derecha a izquierda (en vez de colocar el mínimo en el primer espacio de la tabla, lo coloca en el último). Discutimos un poco más las similitudes y diferencias entre ambos algoritmos en el apartado 6.6.

> ⚠ **Control de parámetros**:
>
> - Rechazamos los `array == NULL` porque no hay tabla de elementos que ordenar.
> - Rechazamos los `ip < 0` porque un índice negativo para una tabla no tiene sentido (al menos en C).

Pseudocódigo

```
SelectSort(array T, P, U):
  for i from U to P + 1:    # i decreasing
    min = min(T, P, i)
    swap(T[i], T[min])
```

4. Código fuente

4.1 Generación de un número aleatorio (función `random_num()`)

```
int random_num(int inf, int sup) {
  /* Comprobación de parámetros */
  if (inf > sup || inf < 0)
    return ERR;

  return inf + (int) ((sup - inf + 1.0) * (rand() / ((double)RAND_MAX + 1.0)));
}

int random_num_mal(int inf, int sup) {
  return inf + (rand() % (sup - inf + 1));
}
```

4.2 Generación de una permutación aleatoria (`función generate_perm()`)

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

  /* Comprueba parámetros */
  if (N <= 0)
```

```
    return NULL;

  /* Reserva memoria para los números */
  perm = (int *) calloc(N, sizeof(int));
  if (!perm)
    return NULL;

  /* Genera los números */
  for (i = 0; i < N; i++) {
    perm[i] = i + 1;
  }

  /* Permuta los números */
  for (i = 0; i < N; i++) {
    j = random_num(i, N-1);
    aux = perm[i];
    perm[i] = perm[j];
    perm[j] = aux;
  }

  return perm;
}
```

4.3 Generación de permutaciones aleatorias (`función generate_perms()`)

```
int ** generate_permutations(int n_perms, int N) {
  int ** perms, i, flag = OK;

  /* Comprueba parámetros */
  if (n_perms <= 0 || N <= 0)
    return NULL;

  /* Reserva memoria para los punteros a las permutaciones */
  perms = (int **) malloc(n_perms * sizeof(int *));
  if (!perms) {
    return NULL;
  }

  /* Genera las permutaciones */
  for (i = 0; i < n_perms && flag == OK; i++){
    perms[i] = generate_perm(N);
    if (perms[i] == NULL) {
      flag = ERR;
    }
  }

  /* Comprueba errores en la generación de permutaciones */
  if (flag == ERR) {
    for (i -= 2; i >= 0; i--) {
      free(perms[i]);
      free(perms);
      return NULL;
    }
  }

  return perms;
}
```

4.4 Ordenación por selección de permutaciones (`función SelectSort()`)

```
int SelectSort(int* array, int ip, int iu)
{
  int i, minimo, sum = 0;

  /* Comprueba parámetros */
  if (!array || ip < 0)
    return ERR;

  /* Algoritmo */
  for (i = ip; i < iu; i++){
    minimo = min(array, i, iu);
    sum += iu - i;
    swap(array + i, array + minimo);
  }
```

```
  return sum;
}
```

Swap (función privada)

Función auxiliar para intercambiar dos elementos de la tabla.

```
void swap(int *e1, int *e2) {
  int aux;

  /* Comprueba parámetros */
  if (!e1 || !e2 || *e1 == *e2)
    return;

  aux = *e1;
  *e1 = *e2;
  *e2 = aux;
}
```

Min (función privada)

Función auxiliar para calcular el mínimo del array.

```
int min(int* array, int ip, int iu)
{
  int j, minimo;

  /* Comprueba parámetros */
  if (!array || ip < 0)
    return ERR;

  /* Busca el mínimo */
  minimo = ip;

  for (j = ip + 1; j <= iu; j++)
      if (array[j] < array[minimo])
        minimo = j;

  return minimo;
}
```

4.5 Medición de tiempos de ejecución (funciones de `times.h`)

Average sorting time

```
short average_sorting_time(pfunc_sort metodo, int n_perms, int N, PTIME_AA ptime)
{
  int **array, total_ob = 0, i, ob, min_ob, max_ob;
  clock_t comienzo, final;

  /* Comprueba parámetros */
  if (!metodo || n_perms <= 0 || N <= 0 || !ptime)
    return ERR;

  /* Genera las permutaciones */
  array = generate_permutations(n_perms, N);
  if (!array)
    return ERR;

  /* Definimos valores por defecto para OB mínimas y máximas (son límites) */
  min_ob = INT_MAX;
  max_ob = 0;

  /* Comienza el test de rendimiento */
  comienzo = clock();

  for (i = 0; i < n_perms; i++) {
    /* Ordena la permutación i-ésima */
    ob = metodo(array[i], 0, N - 1);
    if (ob == ERR) {
```

```
      free_perms(array, n_perms);
      return ERR;
    }

    /* Vamos contando el número total de OB en todas las permutaciones para calcular después el promedio */
    total_ob += ob;

    /* Actualizamos los valores de OB máxima y mínima (en la primera iteración toman valores con sentido) */
    if (max_ob < ob)
      max_ob = ob;

    if (min_ob > ob)
      min_ob = ob;
  }

  /* Termina el test de rendimiento*/
  final = clock();

  /* Libera las permutaciones */
  free_perms(array, n_perms);

  /* Almacenamos los datos necesarios en la estructura ptime */
  ptime->n_elems = n_perms;
  ptime->N = N;
  ptime->time = (double)(final - comienzo) / (double)n_perms;
  ptime->average_ob = (double)total_ob / (double)n_perms;
  ptime->min_ob = min_ob;
  ptime->max_ob = max_ob;

  return OK;
}
```

Generate sorting times

```
short generate_sorting_times(pfunc_sort method, char* file, int num_min, int num_max, int incr, int n_perms)
{
  PTIME_AA sorting_times;
  int i, j, flag, num_ptimes;

  /* Comprueba parámetros */
  if (!method || !file || num_min <= 0 || num_max < num_min || incr <= 0 || n_perms <= 0)
    return ERR;

  /* Cálculo del número de tamaños a probar */
  num_ptimes = (num_max - num_min) / incr + 1;

  /* Reserva de memoria para las estructuras que almacenan los datos */
  sorting_times = (PTIME_AA) malloc(num_ptimes * sizeof(TIME_AA));
  if (!sorting_times)
    return ERR;

  /* Cálculo de los sorting times para cada tamaño */
  for (i = num_min, j = 0, flag = OK; i <= num_max && flag == OK; i+= incr, j++)
    flag = average_sorting_time(method, n_perms, i, sorting_times + j);

  /* Control de errores */
  if (flag == ERR) {
    free(sorting_times);
    return ERR;
  }

  /* Guarda los sorting times en un fichero */
  flag = save_time_table(file, sorting_times, num_ptimes);
  free(sorting_times);

  return flag;
}
```

Save time table

```
short save_time_table(char *file, PTIME_AA ptime, int n_times)
{
  FILE *pf;
  int i;
```

```
  /* Comprueba parámetros */
  if (!file || !ptime || n_times <= 0)
    return ERR;

  /* Abre el archivo en modo escritura */
  pf = fopen(file, "w");
  if (!pf)
    return ERR;

  /* Imprime los datos del array ptime en el fichero */
  for (i = 0; i < n_times; i++) {
    fprintf(pf, "%d %lf %lf %d %d\n", ptime[i].N, ptime[i].time, ptime[i].average_ob, ptime[i].max_ob, ptime[i].min_ob);
  }

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

Free perms (función privada)

Función auxiliar para liberar las permutaciones.

```
void free_perms(int **array, int num){
  int i;
  for(i = 0; i < num; i++)
    free(array[i]);
  free(array);
}
```

4.6 Ordenación por selección inversa (función `SelectSortInv()`)

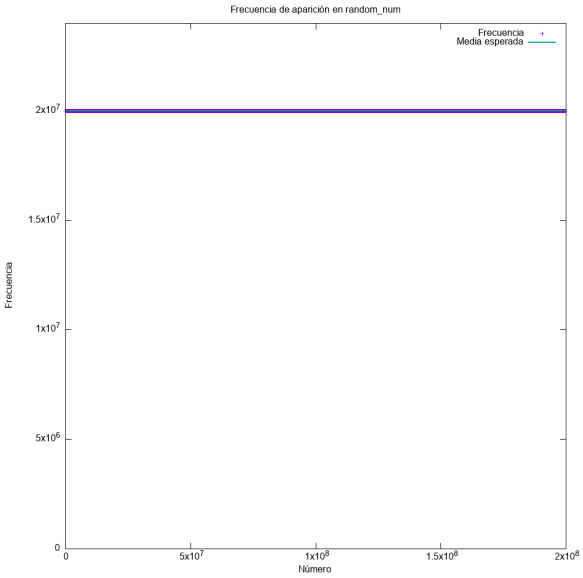
```
int SelectSortInv(int* array, int ip, int iu)
{
  int i, minimo, sum = 0;

  /* Comprueba parámetros */
  if (!array || ip < 0)
    return ERR;

  /* Algoritmo */
  for (i = iu; i > ip; i--){
    minimo = min(array, ip, i);
    sum += i - ip;
    swap(array + i, array + minimo);
  }
  return sum;
}
```

5. Resultados, Gráficas

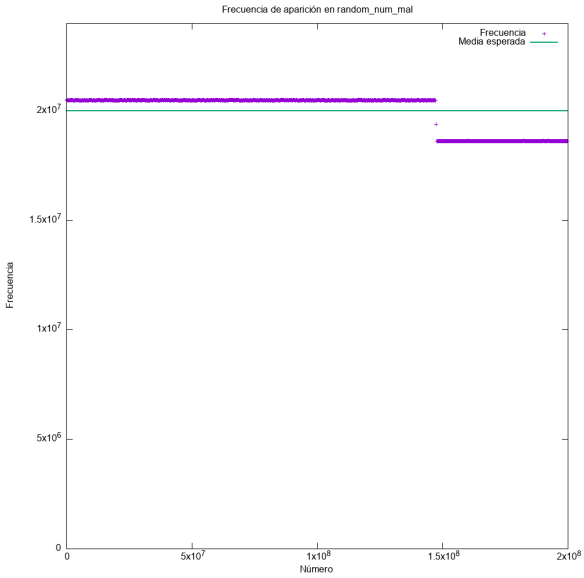
5.1 Generación de un número aleatorio (función `random_num()`)

Hemos graficado los resultados obtenidos al generar 20.000.000.000 de números en un intervalo de amplitud 200.000.000 con incrementos de 200.000:

1. Con nuestra implementación: Nos da una distribución uniforme, teniendo todos los números una frecuencia de aparición similar en torno a la media esperada, es decir, tenemos equiprobabilidad.
2. Con la implementación más rudimentaria (disponible en el apartado 4.1) : La distribución muestra un gran salto de frecuencia en torno al número 14.7300.000. Vemos por tanto que no hay equiprobabilidad (explicamos por qué en el apartado 6.1).



Resultado con la buena implementación



Resultado con la mala implementación

Pruebas

Algunas pruebas básicas para la función:

- Prueba estándar:

```
$ ./exercise1 -limInf 1 -limSup 5 -numN 20
3 5 1 2 3 4 2 1 1 2 2 2 3 4 4 5 3 1 1 3
```

- Prueba con números grandes:

```
$ ./exercise1 -limInf 1 -limSup 2000000 -numN 10
29701 953745 1005350 891064 1941604 96391 1680251 1991407 1886049 407685
```

- Prueba con límites negativos (o por encima de `INT_MAX`, que tiene el mismo efecto):

```
$ ./exercise1 -limInf -3 -limSup 5 -numN 5
-1 -1 -1 -1 -1
```

  (ante error la función devuelve -1)

- Prueba con límites invertidos:

```
$ ./exercise1 -limInf 5 -limSup 1 -numN 5
-1 -1 -1 -1 -1
```

  (ante error la función devuelve -1)

Todos los resultados son los esperados.

Errores y fugas de memoria

Tenemos una función libre de errores y fugas de memoria, como nos indica `valgrind`.

```
$ valgrind --leak-check=full ./exercise1 -limInf 1 -limSup 1000 -numN 4000
...
491
52
694
```

```
486
238
783
209
392
785
...
==7463== All heap blocks were freed -- no leaks are possible
==7463==
==7463== For lists of detected and suppressed errors, rerun with: -s
==7463== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

5.2 Generación de una permutación aleatoria (función `generate_perm()`)

La función `generate_perm()` funciona correctamente, como podemos comprobar en las pruebas realizadas.

Equiprobabilidad

Para la primera posición hay una probabilidad de $\frac{1}{n}$ de que un número determinado acabe en esa posición. Para la segunda posición, cada número tiene una probabilidad de $\frac{n-1}{n}$ de no ser elegido para la primera posición, y en tal caso tiene una probabilidad de $\frac{1}{n-1}$ de ser elegido para la segunda. Por lo tanto la probabilidad es $\frac{n-1}{n} \cdot \frac{1}{n-1} = \frac{1}{n}$.

En el caso general la probabilidad queda así, que como podemos comprobar es equiprobable para todo n.

$$p_m = \frac{n-m+1}{n} \cdot \frac{1}{n-m+1} = \frac{1}{n}$$

Pruebas

Algunas pruebas básicas para la función:

- Prueba estándar:

```
$ ./exercise2 -size 10 -numP 4
9 6 3 7 10 5 4 2 8 1
4 1 7 2 5 10 8 6 9 3
4 5 3 1 7 6 8 2 9 10
9 10 4 2 7 6 3 1 5 8
```

- Prueba con permutación grande:

```
$ ./exercise2 -size 1000000 -numP 1
547795 643972 666843 752140 991233 167254 142162 39342 773508 820390 972487 565545 53160 ...
```

- Prueba con tamaño mayor que `INT_MAX` (o negativo, que tiene el mismo efecto):

```
$ ./exercise2 -size 300000000000 -numP 1
Error: Out of memory
```

- Prueba con número de permutaciones negativo:

```
$ ./exercise2 -size 1000000 -numP -3
```

Todos los resultados son los esperados.

Errores y fugas de memoria

Tenemos una función libre de errores y fugas de memoria, como nos indica `valgrind`.

```
$ valgrind --leak-check=full ./exercise2 -size 10 -numP 1000
...
1 7 8 4 9 2 3 10 5 6
4 6 8 5 9 7 3 10 1 2
```

```
1 7 6 2 8 10 3 5 9 4
6 4 1 5 7 3 9 10 2 8
7 5 6 3 8 4 9 10 2 1
7 10 6 4 1 3 2 8 9 5
...
==7448== All heap blocks were freed -- no leaks are possible
==7448==
==7448== For lists of detected and suppressed errors, rerun with: -s
==7448== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

5.3 Generación de permutaciones aleatorias (función `generate_perms()`)

Gracias a las pruebas realizadas podemos afirmar el correcto funcionamiento de `generate_perms()`.

Pruebas

- Prueba estándar:

```
$ ./exercise3 -size 10 -numP 5
3 2 8 4 5 10 7 6 1 9
8 3 1 7 10 6 2 4 5 9
8 9 1 5 10 6 2 4 3 7
8 10 6 3 5 1 4 7 9 2
1 2 6 9 7 5 10 3 4 8
```

- Prueba con permutación grande:

```
$ ./exercise3 -size 1000000 -numP 1
169625 716247 529723 650564 884506 629510 241492 174744 800743 788927 ...
```

- Prueba con tamaño mayor que `INT_MAX` (o negativo, que tiene el mismo efecto):

```
$ ./exercise2 -size 300000000000 -numP 1
Error: Out of memory
```

- Prueba con número de permutaciones negativo:

```
$ ./exercise2 -size 1000000 -numP -3
Error: Out of memory
```

Todos los resultados son los esperados.

Errores y fugas de memoria

Además, tenemos una función libre de errores y fugas de memoria, como nos indica `valgrind`.

```
$ valgrind --leak-check=full ./exercise3 -size 10 -numP 1000
...
3 9 10 7 5 2 8 1 6 4
5 4 7 10 1 3 8 9 2 6
8 7 1 9 4 6 3 2 10 5
5 10 8 6 7 9 1 2 3 4
9 3 6 2 4 8 1 7 5 10
7 8 3 6 2 10 5 4 9 1
8 6 10 7 3 4 2 1 5 9
2 9 3 8 6 7 5 10 4 1
5 4 3 7 8 1 6 10 2 9
10 4 1 9 2 3 7 6 8 5
9 2 10 3 1 6 7 4 5 8
2 5 3 4 10 9 6 8 1 7
8 6 7 9 3 5 4 2 1 10
...
==7469== All heap blocks were freed -- no leaks are possible
==7469==
==7469== For lists of detected and suppressed errors, rerun with: -s
==7469== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

5.4 Ordenación por selección de permutaciones (función `SelectSort()`)

Podemos afirmar que el algoritmo de ordenación $SelectSort$ funciona adecuadamente ordenando permutaciones aleatorias, como podemos comprobar en las pruebas realizadas.

Pruebas

Algunas pruebas básicas para la función:

- Prueba estándar:

```
$ ./exercise4 -size 20
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

- Prueba con tamaño muy grande:

```
$ ./exercise4 -size 100000
1 2 3 4 5 6 7 8 9 ... 99993 99994 99995 99996 99997 99998 99999 100000
```

- Ordenar solo la segunda mitad de lista: modificamos `exercise4.c`, cambiando la llamada a la función por `SelectSort(perm, tamano/2, tamano-1)`

```
$ ./exercise4 -size 20
permutación sin ordenar: 20 13 14 6 4 7 18 16 12 15 8 19 17 10 9 5 11 3 2 1
permutación ordenada: 20 13 14 6 4 7 18 16 12 15 1 2 3 5 8 9 10 11 17 19
```

  Los resultados son los esperados (ordena lo que le pedimos).

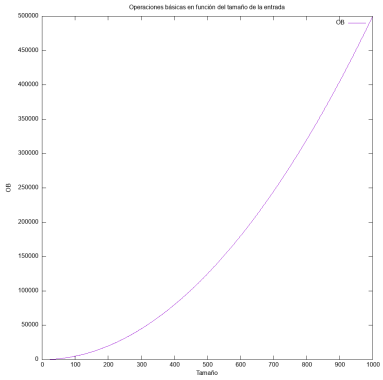
Errores y fugas de memoria

Tenemos una función libre de errores y fugas de memoria, como nos indica `valgrind`.

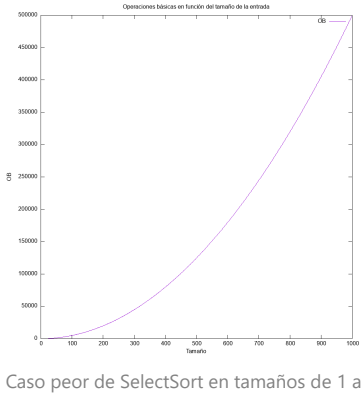
```
$ valgrind --leak-check=full ./exercise4 -size 1000
...
1       2       3       4       5       6       7       8       9       10
11      12      13      14      15      16      17      18      19      20
21      22      23      24      25      26      27      28      29      30
31       32      33      34      35      36 ...
...
==7840== All heap blocks were freed -- no leaks are possible
==7840==
==7840== For lists of detected and suppressed errors, rerun with: -s
==7840== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

5.5 Medición de tiempos de ejecución (funciones de `times.h`)

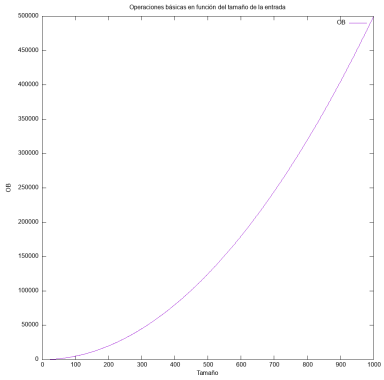
Para $SelectSort$ observamos que el caso mejor, peor y medio son idénticos, como esperábamos, pues el algoritmo no distingue el grado de desorden de la entrada. En todos los casos el rendimiento es cuadrático: $\frac{n^2}{2} - \frac{n}{2}$.



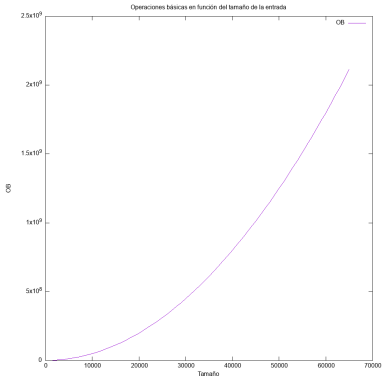
Caso mejor de SelectSort en tamaños de 1 a 1000 con incrementos de 1



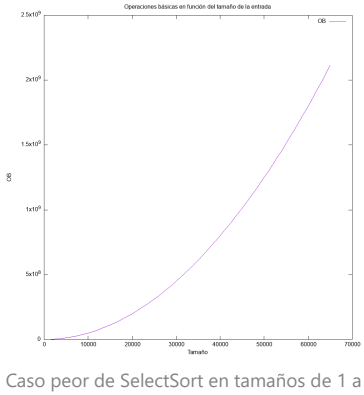
Caso peor de SelectSort en tamaños de 1 a 1000 con incrementos de 1



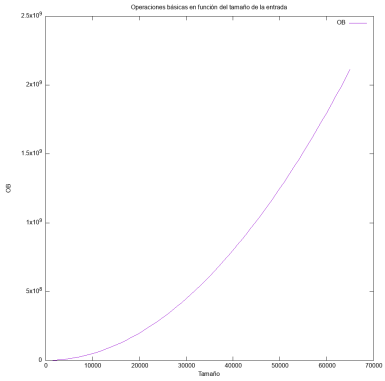
Caso medio de SelectSort en tamaños de 1 a 1000 con incrementos de 1



Caso mejor de SelectSort en tamaños de 1 a 65001 con incrementos de 1000

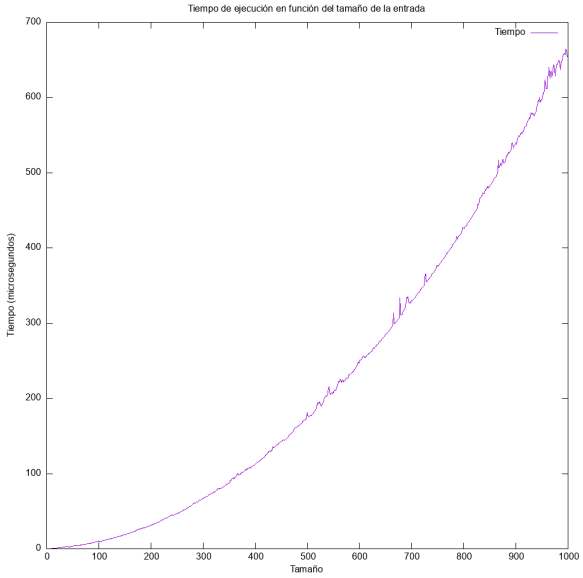


Caso peor de SelectSort en tamaños de 1 a 65001 con incrementos de 1000

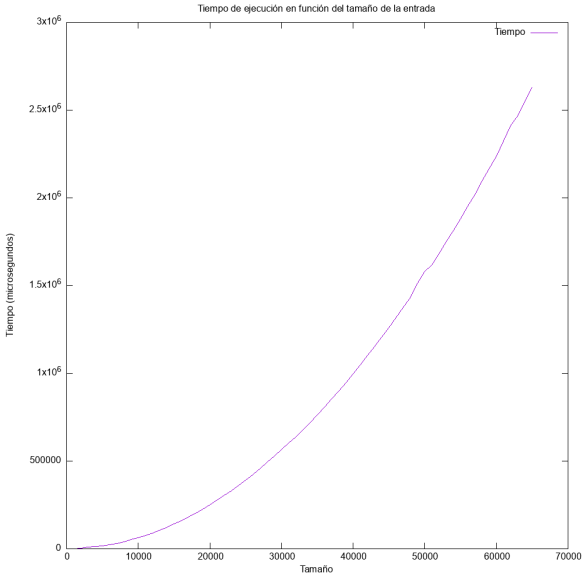


Caso medio de SelectSort en tamaños de 1 a 65001 con incrementos de 1000

El tiempo medio de reloj se ajusta bastante al TAE calculado en operaciones básicas. Sin embargo, en la gráfica con más detalle aparece una cierta distorsión debido a las circunstancias de la ejecución: por ejemplo, si el procesador estaba ocupado durante la prueba para el tamaño 700, se puede producir un retraso y que el tiempo real sea superior al que correspondería teóricamente. Esto es algo que cabe esperar de la ejecución física del algoritmo en una máquina.



Tiempo medio de ejecución de SelectSort en tamaños de 1 a 1000 con incrementos de 1



Tiempo medio de ejecución de SelectSort en tamaños de 1 a 65001 con incrementos de 1000

Errores y fugas de memoria

Tenemos funciones libres de errores y fugas de memoria, como nos indica `valgrind`.

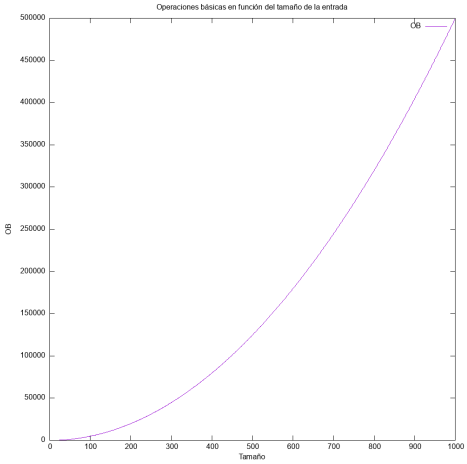
```
$ valgrind --leak-check=full ./exercise5 -num_min 1 -num_max 1000 -incr 1 -numP 10 -outputFile stats5_valgrind.txt
...
Correct output
...
==7536== All heap blocks were freed -- no leaks are possible
==7536==
==7536== For lists of detected and suppressed errors, rerun with: -s
==7536== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

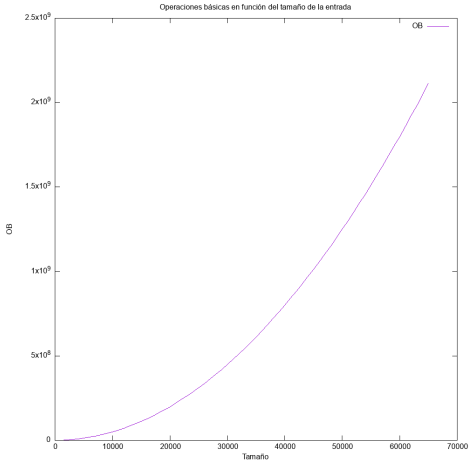
Como se puede observar en el siguiente fichero de texto, en las dos primeras líneas hay un aumento considerable de tiempo en comparación con las siguientes: el tamaño 3 se ordena tan rápido que no pasa ni un tic de reloj, pero hay retrasos en los tamaños 1 y 2 que hacen que tarden más. Esto se puede atribuir a las comprobaciones que hace `valgrind` dentro de la función de ordenación. Como hemos dicho, los tiempos reales de ejecución dependen de las circunstancias en que se ejecuta el algoritmo.

```
Num   Tiempo       OB_media    OB_min  OB_max
1     87.000000    0.000000    0       0
2     101.400000   1.000000    1       1
3     1.100000     3.000000    3       3
4     1.200000     6.000000    6       6
5     1.400000     10.000000   10      10
6     1.800000     15.000000   15      15
7     2.200000     21.000000   21      21
8     2.700000     28.000000   28      28
9     3.200000     36.000000   36      36
10    5.900000     45.000000   45      45
11    8.600000     55.000000   55      55
12    5.800000     66.000000   66      66
13    6.300000     78.000000   78      78
14    6.900000     91.000000   91      91
15    7.500000     105.000000  105     105
16    10.000000    120.000000  120     120
17    11.800000    136.000000  136     136
18    13.000000    153.000000  153     153
19    17.300000    171.000000  171     171
20    13.300000    190.000000  190     190
...
```

5.6 Ordenación por selección inversa (función `SelectSortInv()`)

Para $SelectSortInv$ observamos de nuevo que el caso mejor, peor y medio son idénticos, como esperábamos, pues el algoritmo no distingue el grado de desorden de la entrada. De hecho, también coinciden las gráficas con las de $SelectSort$, pues el rendimiento de ambas es el mismo: $\frac{n^2}{2} - \frac{n}{2}$.

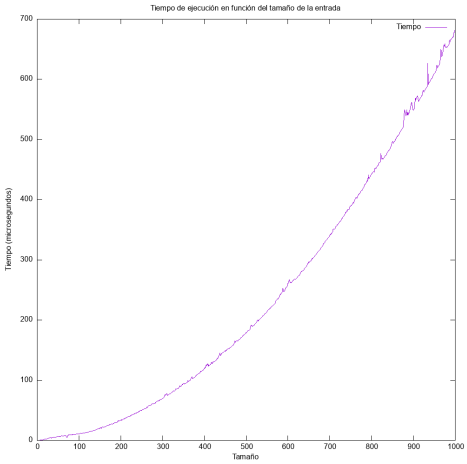




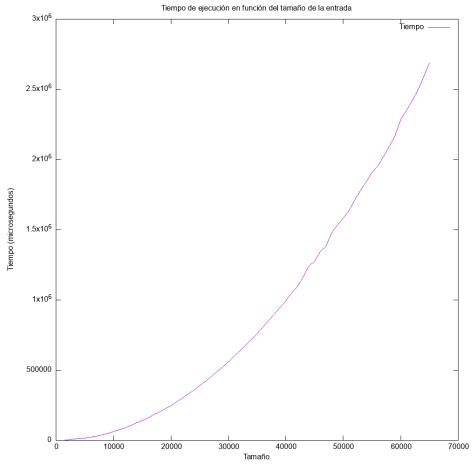
TAE (mejor, peor y medio) de SelectSortInv en tamaños de 1 a 1000 con incrementos de 1

TAE (mejor, peor y medio) de SelectSortInv en tamaños de 1 a 65001 con incrementos de 1000

Como en el caso de su gemelo $SelectSort$, el tiempo puro de ejecución de $SelectSortInv$ se ajusta bastante al TAE calculado en operaciones básicas, obviando las pequeñas distorsiones que pueden ocurrir por las prioridades de procesos en el ordenador.

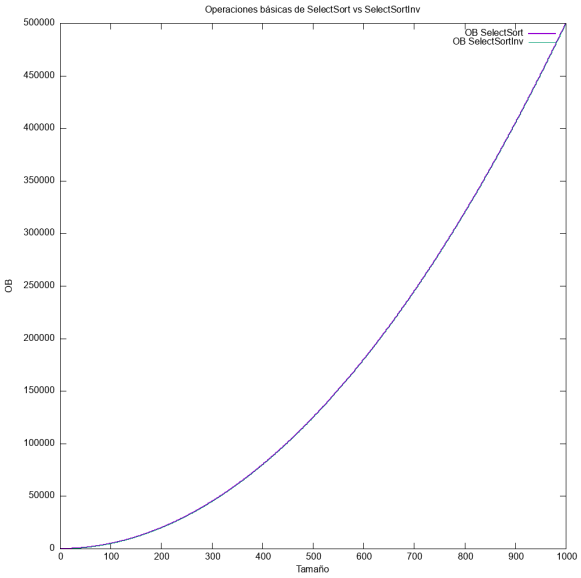


Tiempo medio de ejecución de SelectSortInv en tamaños de 1 a 1000 con incrementos de 1

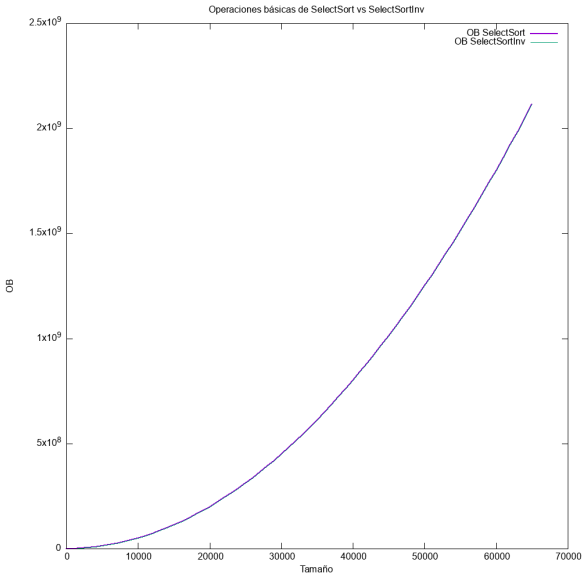


Tiempo medio de ejecución de SelectSortInv en tamaños de 1 a 65001 con incrementos de 1000

Las gráficas de los TAE de ambas funciones se solapan completamente si las ponemos en el mismo lienzo, como cabe esperar ya que los algoritmos funcionan igual y tienen el mismo coste teórico.

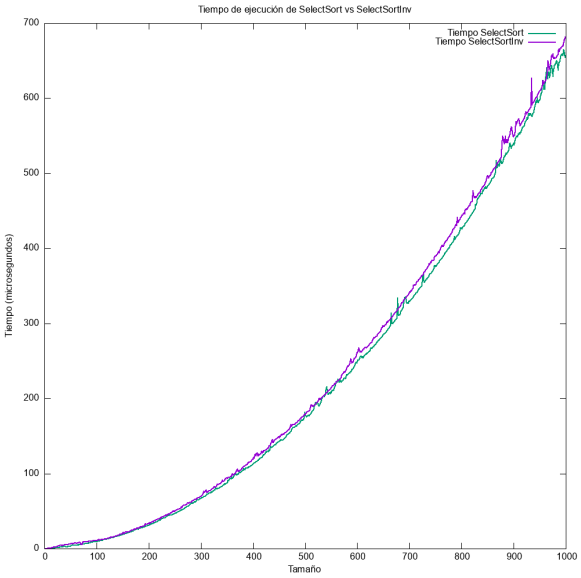


TAE en operaciones básicas de SelectSort vs SelectSortInv en tamaños de 1 a 1000 con incrementos de 1

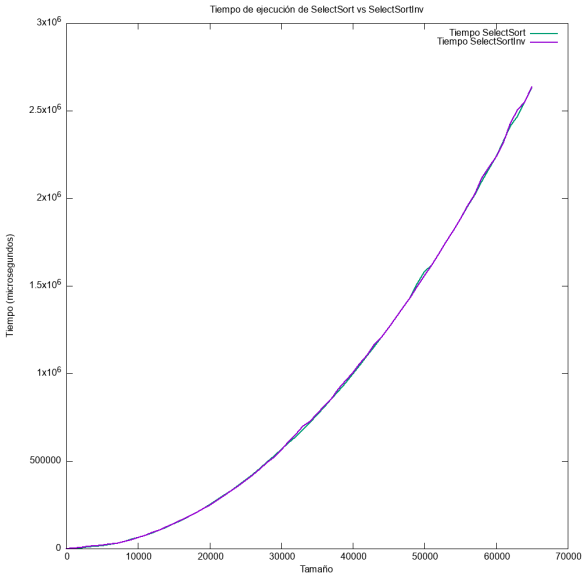


TAE en operaciones básicas de SelectSort vs SelectSortInv en tamaños de 1 a 65001 con incrementos de 1000

A su vez, si nos fijamos en el tiempo puro de ejecución, también siguen el mismo patrón, con desviaciones mínimas por los motivos que ya hemos mencionado (circunstancias de la ejecución de cada uno).



Tiempo medio de ejecución de SelectSort vs SelectSortInv en tamaños de 1 a 1000 con incrementos de 1



Tiempo medio de ejecución de SelectSort vs SelectSortInv en tamaños de 1 a 65001 con incrementos de 1000

Pruebas

Algunas pruebas básicas para la función, hemos modificado el ejercicio 4 para ejecutar la función $SelectSortInv$:

- Prueba estándar:

```
$ ./exercise4 -size 20
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
```

- Prueba con tamaño muy grande:

```
$ ./exercise4 -size 100000
100000 99999 99998 99997 99996 ... 9 8 7 6 5 4 3 2 1
```

- Ordenar solo la segunda mitad de lista: modificamos `exercise4.c`, cambiando la llamada a la función por `SelectSortInv(perm, tamano/2, tamano-1)`

```
$ ./exercise4 -size 20
permutación sin ordenar: 6 5 3 11 14 10 9 16 19 12 20 15 1 4 18 7 2 17 8 13
permutación ordenada: 6 5 3 11 14 10 9 16 19 12 20 18 17 15 13 8 7 4 1 2
```

  Los resultados son los esperados (ordena lo que le pedimos).

Errores y fugas de memoria

Tenemos una función libre de errores y fugas de memoria, como nos indica `valgrind`. Para esta ejecución hemos modificado el `ejercicio4.c` para que use `SelectSortInv()` en lugar de `SelectSort()`.

```
$ valgrind --leak-check=full ./exercise4 -size 1000
...
1000    999     998     997     996     995     994     993     992     991
990     989     988     987     986     985     984     983     982     981
980     979     978     977     976     975     974     973     972     971
970      969     968     967     966     965     964 ...
...
==7875== All heap blocks were freed -- no leaks are possible
==7875==
==7875== For lists of detected and suppressed errors, rerun with: -s
==7875== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

6. Respuesta a las preguntas teóricas

6.1 Pregunta 1

Tras consultar el libro *"Numerical recipes in C"*, hemos implementado nuestra función `random_num()` buscando la mayor equiprobabilidad posible con la siguiente fórmula:

```
inf + (int) ((sup - inf + 1.0) * (rand() / ((double)RAND_MAX + 1.0)))
```

Primero generamos un número aleatorio entre el intervalo $[0, 1)$ con `rand() / ((double) RAND_MAX + 1.0))`. El `+ 1.0` es necesario para que el número resultante nunca pueda ser uno, porque cabe la remota posibilidad de que el output de `rand()` sea `RAND_MAX`. Después, lo multiplicamos por la amplitud del rango de generación `sup - inf + 1.0` y le sumamos el resultado al límite inferior. Con esto conseguimos una función relativamente sencilla.

Otras aproximaciones al problema que no hemos considerado adecuadas son:

- Utilizar el módulo, con una fórmula como esta:

  ```
  inf + rand() % (sup - inf + 1)
  ```

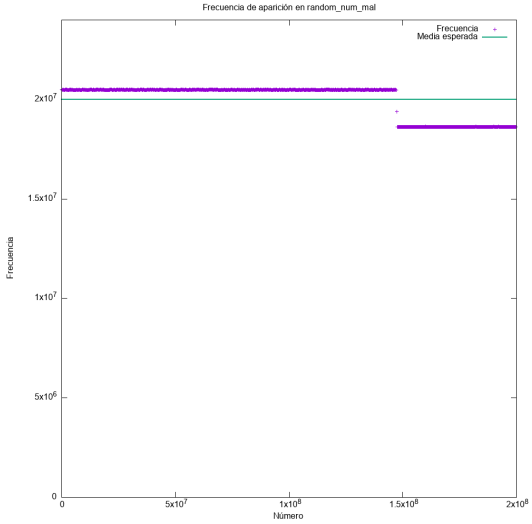
  Pero esta función no es equiprobable cuanto más grande haces tu intervalo, para que sea más fácil la comprensión supongamos que `RAND_MAX = 14`, eso quiere decir que rand() genera números entre el 0 y el 14; y nosotros por ejemplo queremos generar números del 0 al 9. Los posibles resultados son los siguientes:

  - 0 % 10 → 0
  - 1 % 10 → 1
  - 2 % 10 → 2
  - 3 % 10 → 3
  - 4 % 10 → 4
  - 5 % 10 → 5
  - 6 % 10 → 6
  - 7 % 10 → 7
  - 8 % 10 → 8
  - 9 % 10 → 9
  - 10 % 10 → 0
  - 11 % 10 → 1
  - 12 % 10 → 2
  - 13 % 10 → 3
  - 14 % 10 → 4

  Por lo tanto los números del 0 al 4 tienen una probabilidad de 2/15 de aparecer, es decir un porcentaje del 13,33%, mientras que los números del 5 al 8 tienen una probabilidad de 1/15, es decir, un porcentaje del 6,67%. Evidentemente, no es equiprobable.

  Esta función es pseudo-equiprobable si queremos valores aleatorios relativamente pequeños, pero si trabajamos con términos del orden de millones, el defecto es evidente.

La siguiente tabla es la representación de la generación de 20.000.000.000 números aleatorios agrupados en 1000 intervalos de 200.000 números. Es decir, generamos números del 0 al 200.000.000 y necesariamente los contabilizamos en intervalos para tener un archivo manejable. Como podemos observar, no es una función adecuada, ya que en números suficientemente grandes, la equiprobabilidad no está garantizada.



- Utilizar la función `clock()` en lugar de `rand()`:

  Una manera de generar un número aleatorio es utilizar la cantidad de tics del procesador, en C dado por la función `clock()`, pero dicho número no es equiprobable ya que los dígitos de menos peso cambian con mucha más frecuencia que los de mayor peso.

  ### Ejemplo

Dígito	6	5	4	3	2	1
Número de veces que cambia por segundo	1	10	100	1000	10000	100000

6.2 Pregunta 2

En la primera iteración, el algoritmo busca en la tabla $T[P:U]$ el elemento mínimo, y lo intercambia por el primero, $T[P]$. Así, todos los elementos que quedan a la derecha del primero son mayores que este:

$$\forall i > P,\ T[P] < T[i]$$

En las sucesivas iteraciones, se repite el proceso considerando la subtabla que no tiene el primer elemento de la anterior. El mínimo de cada subtabla $T[P+i-1:U]$ va quedando en la posición $T[P+i-1]$. Así la tabla se va ordenando de izquierda a derecha hasta que llegamos a considerar la tabla con únicamente el último elemento, $T[U:U]$, que es el máximo de la tabla total, y termina la ejecución.

6.3 Pregunta 3

Porque cuando $i = U$, estaríamos considerando una subtabla de un único elemento, pero un único elemento siempre está ordenado, así que no es necesario hacer nada.

6.4 Pregunta 4

La comparación de claves (CDC). En esta implementación, se realiza dentro de la función auxiliar `min()`, en la expresión `array[j] < array[minimo]`.

6.5 Pregunta 5

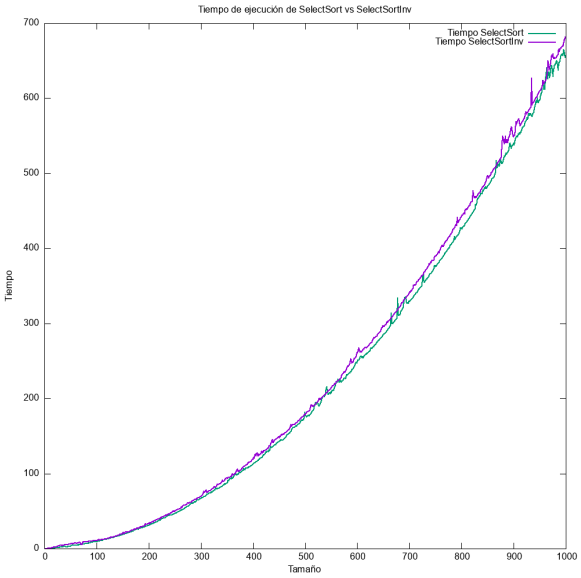
$SelectSort$ es un algoritmo cuyo coste no depende del grado de desorden de la entrada. Es decir, sus casos mejor, peor y medio tienen exactamente el mismo TAE:

$$W_{SS}(n) = B_{SS}(n) = A_{SS}(n) = \frac{n^2}{2} - \frac{n}{2} = \frac{n^2}{2} + O(n)$$

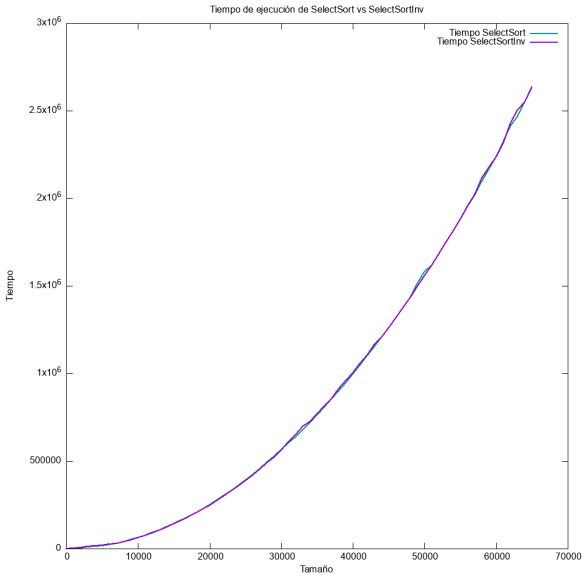
Por este motivo no es un algoritmo muy útil en la aplicación, más allá del enfoque académico y didáctico.

6.6 Pregunta 6

Si comparamos los algoritmos $SelectSort$ y $SelectSortInv$, podemos observar que a efectos prácticos el tiempo puro de ejecución es prácticamente el mismo (el TAE es idéntico). Concuerda con lo esperado, ya que ejecuta casi el mismo código, solo que posicionándolo al revés en el array, utilizando las mismas funciones auxiliares y estrategias.



Tiempo de SelectSort vs SelectSortInv en tamaños de 1 a 1000 con incrementos de 1



Tiempo de SelectSort vs SelectSortInv en tamaños de 1 a 65001 con incrementos de 1000

7. Conclusiones finales

En esta práctica hemos diseñado las herramientas necesarias para analizar cualquier algoritmo de ordenación, no solo $SelectSort$, enfocándonos en el análisis de rendimiento, más allá de la implementación concreta de nuestras funciones. La librería `time.h` será muy útil en prácticas posteriores para medir los tiempos de ejecución, y hemos aprendido a manejar Gnuplot para mostrar los resultados en gráficas de forma visual.

En cuanto al algoritmo concreto analizado, consideramos que no es realmente útil para la ordenación por su elevado coste. Además, ni siquiera podríamos buscarle un caso de uso concreto como a otros algoritmos de ordenación locales, pues el coste es constante independientemente del grado de desorden de la entrada.