Análisis de Algoritmos 2022/2023

Práctica 1

David Brenchley Uriol, Javier San Andrés de Pedro, 1201.

Código	Gráficas	Memoria	Total

1. Introducción.

En esta práctica, hemos trabajando con los algoritmos de ordenación cuadrática y hemos puesto en práctica de manera experimental las hipótesis aventadas en teoría. Además, nos hemos familiarizado con herramientas como GNUPlot y diferentes algoritmos famosos para generar permutaciones equiprobables como es el algoritmo Fischer-Yates. Además, hemos enfrentado la problemática para generar números completamente aleatorios con un ordenador.

2. Objetivos

2.1 Generación de números aleatorios en un intervalo

Familiarizarse con la problemática de generación de números arbitrarios con un ordenador, así como la dificultad para generar distribuciones equiprobables en un intervalo determinado (conflicto con la congruencia en un determinado módulo).

2.2 Generar permutaciones aleatorias de números de 1 a N

Mediante la utilización del apartado anterior, generar una permutación aleatoria dentro de un array de orden N. Algoritmo Fischer-Yates y su equiprobabilidad.

2.3 Generación de múltiples permutaciones aleatorias de 1 a N

Aplicar el algoritmo del apartado anterior pero varias veces para poder observar, de hecho, que el algoritmo anterior genera distribuciones equitativas de probabilidad.

2.4 SelectSort

Utilizar un algoritmo local de ordenación (cuadrático en este caso) y asentar los conceptos de operación básica vistos en teoría.

2.5 Medición experimental de tiempos

Evaluar, mediante un aparato de medida de tiempo (protagonizado en este caso por la biblioteca time.h), el coste a nivel de tiempo de un determinado algoritmo de ordenación cualquiera que cumpla con el prototipo de función que se especifica en el programa en sí. En este caso, evaluaremos y realizaremos gráficas sobre el SelectSort.

2.6 SelectSortInv

Modificación del SelectSort usual y su correspondiente aplicación del anterior apartado.

3. Herramientas y metodología

La práctica se ha realizada en *Linux*, utilizando *Visual Studio* como herramienta para la programación del código correspondiente. Para la depuración de dicho código y el control de la reserva y fuga de memoria, se ha recurrido a *gcc* y *valgrind*. Finalmente, para la visualización de los resultados impresos por pantalla, se ha hecho uso se *sort* y *uniq*, además de *gnuplot* para la realización de gráficas.

En esta sección, comentaremos la metodología llevada a cabo para la resolución de los diferentes problemas y su correspondiente solución. Por otro lado, comentamos de manera breve las entradas y salidas de cada una de las rutinas; aunque para especificaciones más concretas, cada función viene bien documentada en los ficheros que contienen los códigos fuentes (por eso no consideramos necesario explayarnos en ello).

3.1 Generación de números aleatorios en un intervalo

Entrada: int inf, int sup (que vienen a marcar el intervalo de donde se quiere sacar un número aleatorio).

Salida: número entero aleatorio entre inf y sup.

El principal reto de este ejercicio es conseguir lidiar las deficiencias de utilizar congruencias módulo n para generar aleatoridad en el intervalo. Nuestra primera idea, la primera que te surge sin darle muchas más vueltas; es restringir la generación de números aleatorios al anillo Z_n (sería un cuerpo si n fuera primo), donde n = sup-inf+1. Es decir, utilizar el comando: rand() % (sup-inf+1). Esto lo que hace es modular el número aleatorio que sale de utilizar rand(). Luego bastaría sumar a ese número el ínfimo para obtener un número aleatorio en el intervalo que se nos pide. No obstante, ese método lo descartamos posteriormente al darnos cuenta del siguiente contraejemplo:

Si, por ejemplo, queremos generar números aleatorios entre el 0 y el 2, siendo estos nuestra cota inferior y superior respectivamente. De acuerdo a lo anterior, n=3. El problema está en que si por ejemplo RAND_MAX fuera 10, entonces hay 4 números congruentes con 0 (0, 3, 6, 9), otros cuatro congruentes con 1 (1, 4, 7, 10); pero solo habría tres números congruentes con 2 (2, 4, 8). Por tanto, el número 2 tendría menos probabilidades de aparecer que los anteriores, mermando la equiprobabilidad.

Obviamente, RAND_MAX suele ser un número más grande, pero para que haya equiprobabilidad debería ser múltiplo de todos los números por debajo de él (cosa que es no solo imposible, sino inviable).

Otra aproximación con la que tratamos de acercarnos a resolver el problema fue adaptar el RAND_MAX multiplicando por (sup-inf+1) y restando 1. Es decir, con la notación de n antes referida, n*RAND_MAX-1. El código pasaría a ser: (n*rand() - 1) % n. Tras hacer la demostración matemática, se puede comprobar que, en ese intervalo, las clases de equivalencia módulo n tienen todas el mismo cardinal. El problema de este método viene cuando el intervalo es tan grande que superamos INT_MAX.

Por último, optamos por recurrir a la lectura del libro y se nos plantean, esencialmente, dos métodos a seguir. A continuación, explicaremos de manera breve nuestra elección y luego, en la cuestión 1, entraremos más en profundidad.

En el libro se nos ofrecen, realmente, unos cuantos métodos de ordenación. De ellos, podemos destacar principalmente dos. Nosotros hemos optado por el primero que se da (el que se basa en los bits más significativos). El segundo del libro lo hemos descartado porque, a pesar de ser más rápido, es precisamente el primero en el que habíamos pensado y creemos que nuestras razones para no escogerlo son bastante claras. Aunque hemos de admitir que dado un N lo suficientemente grande, entonces la diferencia es mínima y el otro método tarda menos.

Eligiendo el anterior, lo que hacemos es crear una distribución uniforme de números aleatorios en un intervalo entre 0 (incluido) y 1 (no incluido). A partir de ahí trasladamos mediante un producto) dicha distribución al intervalo donde queremos nuestros números aleatorios.

3.2 Generar permutaciones aleatorias de números de 1 a N

Entrada: int N (número de elementos de la permutación a generar)

Salida: permutación (array de enteros que están permutados entre sí).

Para la resolución de este ejercicio, simplemente nos hemos basado en el pseudocódigo que se nos aporta para su elaboración:

```
para i de 1 a N:
   perm[i] = i;

para i de 1 a N:
   intercambiar perm[i] con perm[random_num(i, N)];
```

Lo primero que toca hacer es hacer reserva de memoria dinámica de un array de enteros de tamaño N (ya que en el pseudocódigo se puede apreciar que este es el tamaño.

No obstante, a la hora de implementar el pseudocódigo en C, cualquier array (perm en este caso) está indexado desde el 0 hasta N-1. Luego en la primera parte, si queremos un array de 1 a N, lo que tenemos que hacer es que perm[i]=i+1 y que i vaya de 0 a N-1.

Otra alternativa sería hacer reserva de memoria de un array de tamaño N+1 y utilizar solo los elementos indexados por el 1 y N, pero esta solución es, simplemente, poco ortodoxa ya que se reserva más memoria de la que se necesita.

Por otro lado, para la segunda parte, simplemente tenemos que ser consecuentes con el array creado anteriormente y con lo citado en la primera parte. En nuestro caso, tenemos que ir en un bucle de 0 a N-1 (sea i su índice) e intercambiar cada elemento en la posición i por cualquier otro en una posición arbitraria entre el propio i y N-1. Esa sería la adaptación del pseudocódigo a nuestra redefinición de términos.

Sin embargo, a la hora de implementarlo en C, hay cosas que varían. Para empezar, no tiene sentido ir hasta N-1 ya que en N-1 el número aleatorio entre sí mismo va a ser sí mismo. Por ende, no se va a poder cambiar por nadie. El bucle que generemos solo tiene que ir hasta N-2 (le sobra una interacción al pseudocódigo). Quitando ese aspecto, para generar ese intercambio hemos creado la función void swap (int* x, int* y) que, de hecho, hemos incluido en permutations.h ya que es muy probable que la volvamos a necesitar. Simplemente, le pasamos las direcciones de memoria de dos punteros a entero (en nuestro programa dos punteros a elementos de la tabla) e intercambia dichas direcciones.

Para generar dicha aleatoridad, hemos simplemente recurrido a la función de generación de números aleatorios del apartado anterior. Esa función nos garantiza la aleatoriedad de la permutación, que es al fin y al cabo lo que se persigue.

Este apartado supone la culminación del algoritmo de Fischer-Yates. Todo lo realizado es, en última instancia, la descripción exacta de dicho algoritmo. No obstante, en la publicación que dieron no se reutiliza el mismo array, es decir, en vez de coger e ir almacenando en el propio array, lo continúan con otro array. Se toma un número al azar, se tacha del primer array y se escribe en el siguiente (esto realmente es un ajuste de lo que realmente proponen Fischer y Yates).

La implementación llevada a cabo corresponde más bien con la descripción del algoritmo de Durstenfeld, que es la que usa menos memoria. Es decir, se utiliza el mismo array.

Más información en: https://es.wikipedia.org/wiki/Algoritmo_de_Fisher-Yates

3.3 Generación de múltiples permutaciones aleatorias de 1 a N

Entrada: int n_perms, int N (n_perms delimita el número de permutaciones y N el tamaño de esas permutaciones).

Salida: n_perms permutaciones (n_perms arrays que contienen una permutación de números de 1 a N)

Llevar a cabo este apartado es relativamente sencillo si se han realizado los anteriores. Simplemente, se trata de utilizar la función del apartado anterior para generar, dado un número de permutaciones (n_perms), n_perms permutaciones aleatorias de N elementos. Dicho de otro modo, en este apartado hay que generar un array (de nombre perm_table en nuestro caso) donde cada elemento sea un puntero a un array de permutaciones. En resumidas cuentas, un array de arrays.

En la solución que hemos llevado a cabo, primero creamos ese array de punteros a permutación. Reservamos memoria para crear una tabla de tamaño n_perms cuyos elementos sean punteros a int. Después, para cada elemento de dicha tabla, llamamos a la función del apartado anterior para generar la permutación equiprobable. Esa permutación tiene tamaño N (se especifica en el segundo argumento de la rutina del presente apartado).

Así, obtenemos como resultado (en términos probabilísiticos) un espacio muestral de n_perms de permutaciones todas ellas equiprobables.

3.4 SelectSort

Entrada: int* array, int ip, int iu (array de enteros a ordenar en un intervalo de índices entre ip e iu).

Salida: array ordenado de manera ascendente entre dos índices, número de veces que se realiza la OB.

El algoritmo a implementar es el SelectSort, que es un algoritmo local de ordenación visto en teoría. Para implementarlo, simplemente nos hemos basado en el pseudocódigo siguiente:

```
void SelectSort(Tabla T, ind P, ind U)
        i=P;
        mientras i<U:
          min=i ;
          para j de i+1 a U :
            si T[j]<T[min] :
                min=j ;
          swap(T[i],T[min]) ;
          i++;
```

ırlo, teníamos además que contar el número de veces primera parte es escoger la operación básica. Con el ebe ser CDC (si T[j]<T[min]).

Para implementar la función Select Sort, la parte en que busca el mínimo en la subtabla inmediatamente superior al índice donde nos encontramos, se nos ha instado a efectuarlo mediante una función de cabecera int min (int* array, int ip, int iu, int* ob).

Nosotros hemos implementado esa función, pero consideramos necesario hacer un comentario expresando nuestro desacuerdo con esa implementación. En lugar de pasar cuatro argumentos, se puede desarrollar la misma funcionalidad pasándole tres. Así pues, en lugar de pasarle el array y los índices superior e inferior, podemos pasarle las direcciones de memoria de los elementos del array superior e inferior (dos argumentos) y luego el puntero de la ob.

Por lo demás, en cuanto a implementación, hemos realizado una función para constatar que, efectivamente, el array de salida está ordenado (de manera ascendente). Al final de salida de la función, hay un assert de la función previamente citada. Si por lo que fuera hubiera algún error, ejecutando el programa con las flags correspondientes, se nos notificaría de manera clara.

3.5 Medición experimental de tiempos

En este apartado, hay que realizar varias funciones que se llaman las unas a las otras para, al final, rellenar los campos de una estructura que viene hacer como una ficha técnica de un algoritmo de ordenación.

1. short average_sorting_time

Entrada: pfunc_sort metodo, int n_perms, int N, PTIME_AA ptime (puntero a una función de ordenación, número de permutaciones, tamaño de las permutaciones, puntero a una estructura ptime que lleva un registro del tamaño de las permutaciones, número de permutaciones, veces que se ejecuta la OB media, mínima y máxima; así como tiempo medio de ejecución)

Salida: estructura PTIME_AA rellena

2. short generate_sorting_times

Entrada: pfuncsort method, char *file, int num_min, int num_max, int incr, int n_perms (puntero a función de ordenación, puntero a char que contiene nombre de un fichero donde imprimir los datos, tamaño mínimo y mínimo de las permutaciones a generar, cociente incremental de dichos tamaños, número de permutaciones a generar por cada tamaño especificado).

Salida: fichero que contiene la ficha técnica del algoritmo de ordenación para diferentes tamaños de permutaciones.

3. save_time_table

Entrada: char* file, PTIME_AA ptime, int n_times (puntero a cadena que contiene el nombre del fichero donde imprimir los datos, puntero a un array de estructuras AAtime, número de estructuras).

Salida: imprimir en fichero los datos de las estructuras pasadas como argumento.

En este ejercicio hay una rutina y dos subrutinas. La función principal, que es generate_sorting_times, utiliza a las otras dos funciones para imprimir en un fichero todos los campos de la estructura ptime para diferentes tamaños de permutaciones que distan entre sí un determinado incremento.

Para implementar dicha rutina, lo primero hay que construir las dos. La primera de todas, average_sorting_time es la más importante ya que nos da el tiempo medio de ejecución y el número de OB mínimo, medio y máximo resultante de ordenar n_perms permutaciones de tamaño N. Así pues, lo que hace la función es crear (llamando a la rutina del apartado 3), dichos permutaciones. Seguidamente y es ahí donde ponemos en marcha nuestro reloj, tiene que ordenar todas las permutaciones implementando la función SelectSort del apartado anterior.

Coetáneamente, va calculado el número de OB mínimo, máximo y medio en base al retorno de la función SelectSort. Una vez termina de ordenar todos, paramos el reloj. Si bien se pudiera argumentar que esta forma no es quizá la mejor para medir el tiempo ya que se realizan más operaciones dentro de ese bucle, simplemente indicar que el mayor peso de la duración de ejecución lo tiene la función de ordenación, suponiendo las demás operaciones una dilación superflua.

No hemos considerado utilizar la función clock_gettime(), de mayor precisión, porque ya acumulamos algo de error de antes y no es necesario añadir tanta precisión. De implementarla, entonces mediríamos minuciosamente y de manera estricta cada tiempo de ejecución de la ordenación: en vez de iniciar el reloj, ordenar todos y luego pararlo; sería iniciar el reloj, ordenar uno, pararlo, guardar ese tiempo y así sucesivamente. Sin embargo, no creemos conveniente semejante cantidad de exactitud.

Por lo demás, la rutina principal de este apartado, solo tiene que llamar a la función anterior para que rellene los campos de la estructura para los diferentes tamaños. Después, llama a la función save_time_table que es la que se encarga de imprimirlos en fichero.

El resultado del apartado es un fichero que contiene para diferentes tamaños, los campos de la estructura (citados en el principio de este apartado).

3.6 SelectSortInv

Entrada: int* array, int iu, int ip (puntero del array a ordenar, índices superior e inferior del intervalo del array a ordenar).

Salida: array ordenado de manera descendente entre dos índices, veces que se ejecuta la OB.

Este último apartado, supone lo mismo que el SelectSort, pero ahora hay que ordenar de manera descendente. Otra vez de nuevo, la OB es una CDC (mismo if), utiliza la misma función de min que el SelectSort para lograr su cometido, lo que pasa que, en vez de empezar por el principio, empieza por el final. Es decir, para no tener que implementar una función max (análoga a min), la estrategia que utilizamos es empezar por el final del intervalo e ir ordenando las subtablas entre el principio e i-1 (siendo i el índice del elemento en el que estamos). Buscamos el mínimo en la subtabla y lo intercambiamos, así sucesivamente.

De esa forma, lo que hacemos es aplicar el SelectSort pero del revés, partiendo desde el final. Al igual que el anterior algoritmo, también hemos diseñado una función para comprobar si el array de salida está ordenado según se quiere. Dicha función viene implementada, de manera análoga, con un assert.

4. Código fuente

4.1 Generación de números aleatorios en un intervalo

```
int random_num(int inf, int sup)
{
  assert(0 <= inf);
  assert(inf <= sup);
  assert(sup<RAND_MAX);
  return inf + (int) (sup - inf + 1.0)*(rand() / (RAND_MAX+1.0));
}
```

4.2 Generar permutaciones aleatorias de números de 1 a N

```
void swap(int* x, int* y)
{
  int aux;
  assert(x!=NULL);
  assert(y!=NULL);

  aux =*x;
  *x=*y;
  *y=aux;
}
```

```
int *generate_perm(int N)
{

  int* perm, i;

  assert(N > 0);

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

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


  for (i = 0; i < N-1; i++)
    swap(&perm[i], &perm[random_num(i, N-1)]);

  return perm;
}
```

(Nota: la función swap la hemos incluido en el archivo .h porque es tan básica que la consideramos de uso general).

4.3 Generación de múltiples permutaciones aleatorias de 1 a N

```
int **generate_permutations(int n_perms, int N)
{
  int** perm_table, i;

  assert (n_perms > 0);
  assert (N > 0);

  if((perm_table = (int **)malloc(n_perms*sizeof(perm_table[0])))==NULL)
    return NULL;

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

    if((perm_table[i]=generate_perm(N))==NULL)
    {
      int j;
      for(j=0; j<i; j++)
        free(perm_table[j]);
    }
  }

  return perm_table;
}
```

4.4 SelectSort

```
int min(int* array, int ip, int iu, int* ob)
{
  int i, minimo=ip;

  assert(array!= NULL);
  assert(ob != NULL);

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

    (*ob)++;
  }

  return minimo;
}
```

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

  assert(array!=NULL);
  assert(ip>=0);
  assert(ip<=iu);

  while(i<iu)
  {
    minimo=min(array, i,iu, &ob);
    swap(&array[i],&array[minimo]);
    i++;
  }

  assert(Array_is_Sorted(&array[ip],&array[iu])==OK);

  return ob;
}
```

(Nota: antes de salir de salir de la función, comprobamos que, efectivamente, el array que damos como salida está ordenado, eso sí seleccionamos la flag en el makefile correspondiente).

4.5 Medición experimental de tiempos

```
short average_sorting_time(pfunc_sort metodo, int n_perms, int N, PTIME_AA ptime)
{
  int **permutaciones, i, temp, flag=OK;
  double ini, fin;

  assert(metodo!=NULL);
  assert(ptime!=NULL);
  assert(N>0);
  assert(n_perms>0);


  if((permutaciones=generate_permutations(n_perms,N))==NULL)
    return ERR;

  ptime->N=N;
  ptime->n_elems=n_perms;
  ptime->average_ob=0;
  ptime->max_ob=0;
  ptime->min_ob=INT_MAX;

  ini=clock();
  for(i=0; i<n_perms && flag==OK; i++)
  {
    if((temp=metodo(permutaciones[i],0,N-1))==ERR)
      flag=ERR;

    if(temp < ptime->min_ob)
      ptime->min_ob=temp;

    if (temp > ptime->max_ob)
      ptime->max_ob=temp;

    ptime->average_ob += temp;

  }
  fin=clock();

  if(flag==OK)
  {
    ptime->time=(double)(fin-ini)/CLOCKS_PER_SEC/n_perms;
    ptime->average_ob/=n_perms;
  }

  for(i=0; i<n_perms; i++)
    free(permutaciones[i]);

  free(permutaciones);
  return flag;
}
```

```
short generate_sorting_times(pfunc_sort method, char* file, int num_min, int num_max, int incr,
int n_perms)
{
  PTIME_AA pt;
  int i,j, n_times=((num_max-num_min)/incr)+1;

  assert(file!=NULL);

  if((pt=(PTIME_AA)malloc(n_times*sizeof(TIME_AA)))==NULL)
    return ERR;

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

  save_time_table(file,pt,n_times);
  free(pt);

  return OK;
}
```

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

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

  for(i=0;i<n_times;i++)
  {
    fprintf(f, "    %010d       %011.8f         %013.2f         %010d          %010d      \n",
                ptime[i].N, ptime[i].time,ptime[i].average_ob, ptime[i].max_ob,ptime[i].min_ob);
  }


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

4.6 SelectSortInv

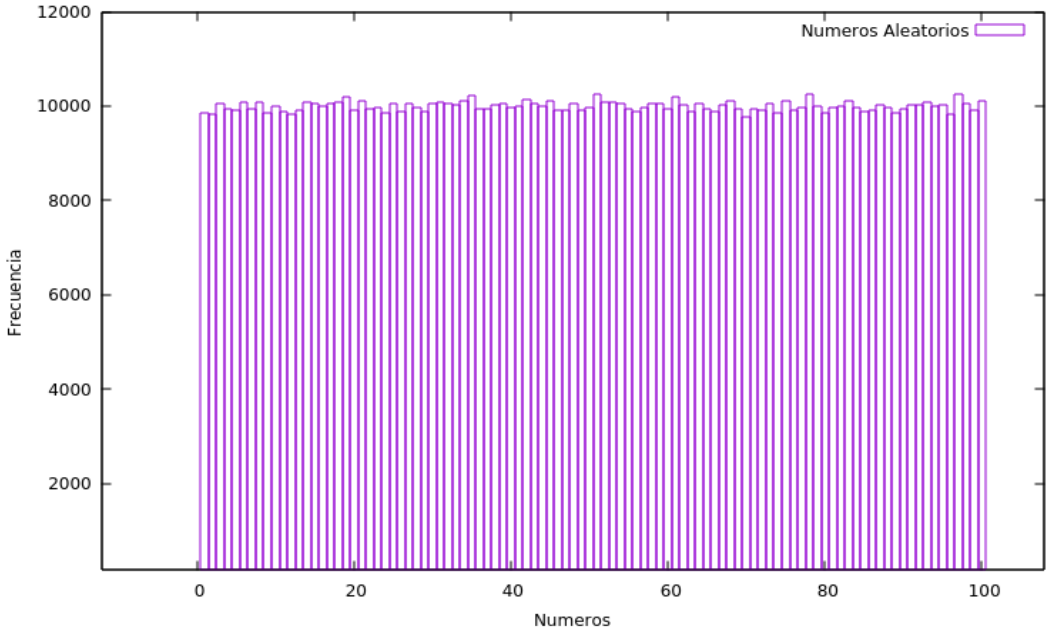
```
int SelectSortInv(int* array, int ip, int iu)
{
  int ob=0, i=iu, minimo;

  assert(array!=NULL);
  assert(ip>=0);
  assert(ip<=iu);

  while(i>ip)
  {
    minimo=min(array, ip,i, &ob);
    swap(&array[i],&array[minimo]);
    i--;
  }

  assert(Array_is_SortedInv(&array[ip],&array[iu])==OK);
  return ob;
}
```

(Nota: la función min es la del apartado anterior).

5. Resultados, Gráficas

1) En el apartado 1 se ha obtenido una función random_num que genera números aleatorios dentro de un rango definido por un ínfimo y un supremo que se pasan como argumento, y se ha comprobado mediante diversas ejecuciones que efectivamente la distribución era equiprobable. Por otro lado, se ha realizado una gráfica representativa de dicha distribución, donde el eje y representa la frecuencia de cada número (eje x).

Grafica Números Aleatorios



Como se puede observar, para una muestra de 1 millón de números cuyo ínfimo es igual a 1 y el supremo igual a 100, la frecuencia absoluta para todos los números es aproximadamente 10000. Luego efectivamente la función random_num genera números aleatorios equiprobables dentro del rango especificado.

2) En el segundo apartado se ha desarrollado int *generate_perm, que devuelve un puntero a una permutación. Para su comprobación se ha procedido a la simulación e impresión por pantalla de permutaciones de diversos tamaños y la creación de una función Array_is_Sorted para comprobar que efectivamente estaban ordenadas. Además de su correspondiente ejercicio (exercise2.c).

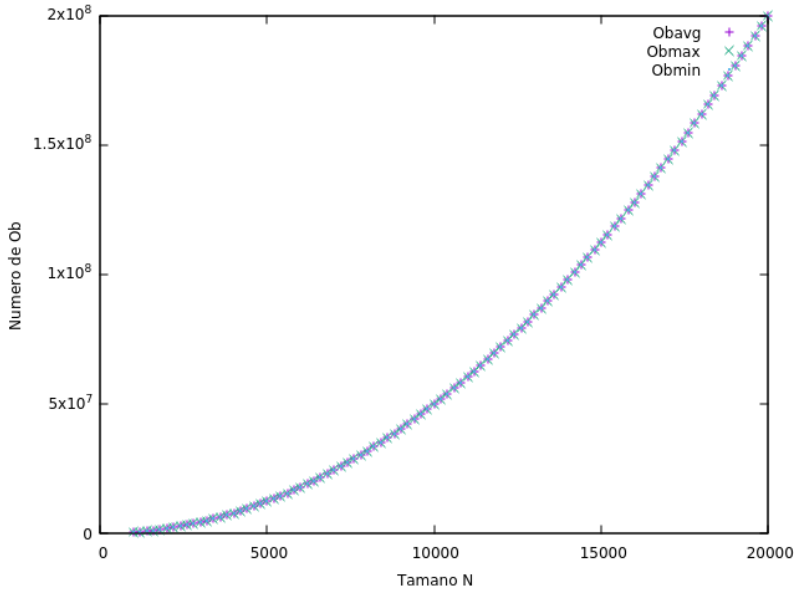
3) En el apartado 3, se ha desarrollado int **generate_permutations que devuelve un puntero a un array de punteros que apuntan a las diversas permutaciones. Su comprobación, similar al caso anterior se basa en la simulación e impresión de

diferentes permutaciones de distintos tamaños, aplicando una metodología muy similar al caso anterior., además de su correspondiente ejercicio (exercise3.c).

4) Para el cuarto apartado se ha desarrollado el algoritmo de ordenación local SelectSort de rendimiento cuadrático, apoyándonos ella función *min.* Dicha función ordena una permutación de menor a mayor. Su comprobación, además del uso de su correspondiente ejercicio (exercise4.c), se ha realizado mediante la comparación de las gráficas obtenidas en el siguiente apartado y el cálculo teórico esperado.

5) Para el apartado 5 se ha implementado generate_sorting_times, junto con save_time_table y average_sorting_time, para poder así ejecutar un algoritmo de comparación, que en nuestro caso es SelectSort, y obtener el número de operaciones básicas y el tiempo de ejecución. Mediante la herramienta de GNU PLOT se han realizado las siguientes gráficas que representan el rendimiento de nuestro algoritmo.

Grafica SelectSortComparacionObs



La grafica SelectSortComparacionObs representada el número de operaciones básicas medias, máximas y mínimas del algoritmo SelectSort para diferentes permutaciones, observándose una evidente función cuadrática idéntica para las 3 mediciones. Esto se ajusta al cálculo teórico del tiempo abstracto de ejecución de SelectSort, pues el algoritmo ejecuta el mismo número de operaciones básicas independientemente del desorden de la permutación inicial.

Además, se puede comprobar que dicho número de operaciones básicas cumple la función:

$$\frac{N^2}{2} - \frac{N}{2} = \frac{N^2}{2} + O(N)$$

donde N representa el tamaño de la permutación que tiene que ordenar.

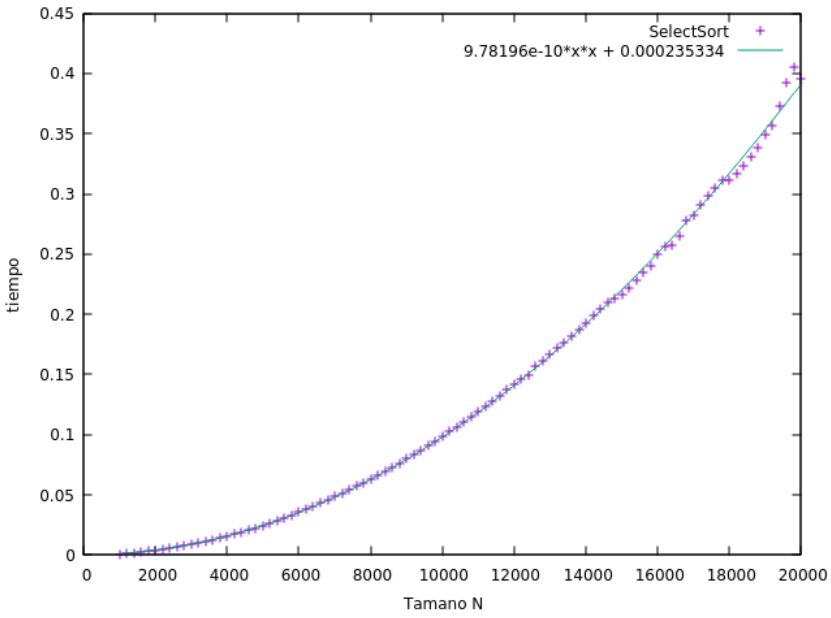
Dicho cálculo se corresponde al número comparaciones que realiza SelectSort, que además coincide con el número máximo de inversiones que tendría que deshacer

En la siguiente imagen se aprecia una porción de los datos de la muestra utilizada para obtener dicha medición, donde la primera fila corresponde al tamaño de la permutación y las tres últimas columnas se corresponden al número de operaciones básicas.

Para el caso N=20000, Ob= $\frac{20000^2}{2} - \frac{20000}{2} = 199990000$ que efectivamente se cumple

0000017600	00.30204900	0154871200.00	0154871200	0154871200
0000017800	00.31053600	0158411100.00	0158411100	0158411100
0000018000	00.31896500	0161991000.00	0161991000	0161991000
0000018200	00.32504700	0165610900.00	0165610900	0165610900
0000018400	00.33828200	0169270800.00	0169270800	0169270800
0000018600	00.35202900	0172970700.00	0172970700	0172970700
0000018800	00.35407000	0176710600.00	0176710600	0176710600
0000019000	00.35284300	0180490500.00	0180490500	0180490500
0000019200	00.36087100	0184310400.00	0184310400	0184310400
0000019400	00.37329600	0188170300.00	0188170300	0188170300
0000019600	00.39327200	0192070200.00	0192070200	0192070200
0000019800	00.38002000	0196010100.00	0196010100	0196010100
0000020000	00.39396600	0199990000.00	0199990000	0199990000

Grafica SelectSortTiempoEjecución



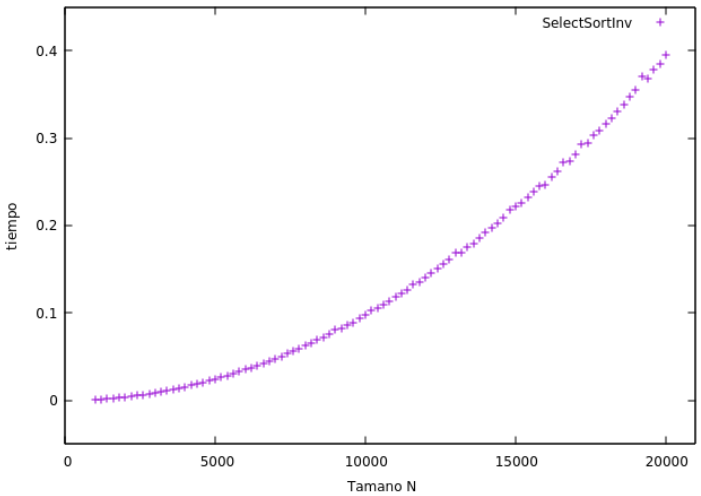
Dicha grafica representa los tiempos de ejecución (en segundos) obtenidos con SelectSort para permutaciones de 1000 hasta 20000 elementos. A dicha representación se le ha añadido una función cuadrática (representada en verde) que se ajusta a la gráfica temporal, lo que corrobora el rendimiento cuadrático del algoritmo.

La próxima tabla de valores muestran los primeros 25 datos de la muestra utilizada, siendo la primera columna el número de elementos y la ultima el tiempo de ejecución (en segundos). Como cabe de esperar, los datos se ajustan a una función cuadrática.

0000001000	00.00120200
0000001200	00.00147800
0000001400	00.00207100
0000001600	00.00272800
0000001800	00.00319300
0000002000	00.00392000
0000002200	00.00531000
0000002400	00.00561100
0000002600	00.00657400
0000002800	00.00791400
0000003000	00.00878300
0000003200	00.00992700
0000003400	00.01115800
0000003600	00.01251800
0000003800	00.01396600
0000004000	00.01544500
0000004200	00.01702200
0000004400	00.01938100
0000004600	00.02148300
0000004800	00.02292900
0000005000	00.02483200

6) Para el último apartado, hemos programado el algoritmo SelectSortInv, y hemos utilizado la misma metodología que con SelectSort, aplicando las mismas comprobaciones que dicho. También hemos comparado el número de operaciones básicas de SelectSortInv con el de SelectSort, y efectivamente coinciden. En la siguiente grafica se muestra el tiempo de ejecución de SelectSortInv, también cuadrática, que se asemeja a la de SelectSort

Grafica SelectSortInvTiempoEjecucion



6. Respuesta a las preguntas teóricas.

Aquí respondéis a las preguntas teóricas que se os han planteado en la práctica.

6.1 Justifica tu implementación de aleat_num ¿en qué ideas se basa? ¿de qué libro/artículo, si alguno, has tomado la idea? Propón un método alternativo de generación de números aleatorios y justifica sus ventajas/desventajas respecto a tu elección.

La idea la hemos tomado del libro *Numerical Recipes In C: the art of scientific computing*. Básicamente, desde el comienzo del libro, nos habla de la dificultad de generar números aleatorios con un ordenador. El problema reside, según se nos comenta, en que puede que un método para generar números arbitrarios consiga equiprobabilidad para un determinado programa; pero sea absolutamente nefasto para cualquier otro.

Nuestra elección se basa en conseguir una desviación homógenea de números entre 0 y el 1. Obviamente, cuanto más homogénea sea nuestra distribución, más paridad habrá. Teniendo en cuenta que (parafraseando al libro) en congruencias modulares, los bits menos significativos son mucho menos aleatorios que los bits menos significativos. Por tanto, nos interesa un método que tenga en cuenta los bits más significativos. Esto lo podemos conseguir mediante una división entre el número aleatorio y RAND_MAX+1. De esta forma, conseguimos un número de tipo float aleatorio entre 0 y 1 (1 no incluido). Para extenderlo a cualquier intervalo de 0 a N simplemente tenemos que multiplicar por N y obtendremos un número de tipo float entre 0 y N. Para acotarlo a un intervalo entre inf y sup simplemente hay que tomar N=sup-inf+1 y sumar inf al número aleatorio que salga de la división y producto anteriores (haciendo su correspondiente casting a int).

Las principales ventajas de mi elección es que consigue mayor aleatoridad. Además, decir que no se ve supeditado a un RAND_MAX, es decir, puede generar números aleatorios en cualquier intervalo que vaya hasta INT_MAX. Sin embargo, es más costoso que el siguiente algoritmo a exponer. Esto es debido a que operar con tipos floats es más costoso que operar con otros tipos de datos.

El otro algoritmo que se puede proponer es un generador de congruencias que trata con bits menos significativos. Este generador pierde bastante aleatoriedad cuando el intervalo es bastante pequeño. No obstante, es bastante rápido y cuando N es lo suficientemente grande genera resultados bastante equiprobables en general.

La idea del algoritmo es generar un número aleatorio cualquiera y modularizarlo en tamaño (sup-inf+1), de tal manera que tendríamos un número aleatorio entre 0 y sup-inf. Después, solo hay que sumarle el inf al anterior número y ya lo tendríamos. (Este algoritmo fue, de hecho, nuestra primera opción).

random_number=inf + rand() % (sup-inf+1).

Otro algoritmo que nació de nuestra cosecha, fue en suplir las deficiencias del anterior algoritmo. Según hemos comentado en la metodología de este apartado, el principal problema es que las clases de equivalencia resultantes de la congruencia modular no tienen todas el mismo cardinal.

Sea n=sup-inf+1, lo que necesitamos para que todas las clases de equivalencia tengan el mismo cardinal es que el número de naturales entre 0 y RAND_MAX sea múltiplo de n. Por tanto, bastaría con reconvertir nuestro RAND_MAX a (n * RAND_MAX -1). El problema es que habría que acceder internamente a la función rand() y cambiarle el límite según convenga. Además, puede haber overflow de manera muy sencilla.

De los mencionados en el libro, voy a destacar uno que, cito textualmente, genera una distribución uniforme de manera rápida y sucia. Sin embargo, es extremadamente rápido. Es el siguiente:

```
unsigned long jran,ia,ic,im;
float ran;
...
jran=(jran*ia+ic) % im;
ran=(float) jran / (float) im;

jran=(jran*ia+ic) % im;
j=jlo+((jhi-jlo+1)*jran)/im;
```

Donde im, ia, ic son coeficientes que hay que seleccionar minuciosamente en una ecuación de recurrencia modular, jran es una semilla para el primer término de dicha sucesión recursiva. El resto jlo sería nuestro inf y jhi nuestro sup.

Por último, hemos comentado que el problema de generar números aleatorios reside en que puede que no funcionen igual de bien para todos los programas. Park y Miller desarrollaron un algoritmo "portable" que se define como algoritmo mínimo de generación de números aleatorios. Ha pasado todos los tests de aleatoridad que se le han realizado. No obstante, el programa no es trivial y no nos merece comentarlo. Dejamos un enlace de una implementación en C:

https://www.sanfoundry.com/c-program-implement-park-miller-random-number-generation-algorithm/

6.2 Justifica lo más formalmente que puedas la corrección (o, dicho de otra manera, por qué ordena bien) del algoritmo SelectSort.

SelectSort ordena bien porque, en cada interacción, va buscando el número mínimo en la siguiente subtabla y lo saca de ahí, de tal manera que ahora hay una subtabla más pequeña de la que saca el mínimo y así sucesivamente.

Dicho de otro modo, SelectSort lo que hace es tomar el primer elemento, buscar el mínimo de la subtabla que resulta al quitarle ese elemento. Después, intercambia ese elemento por el primer elemento que habíamos sacado (si es menor claro). Ahora, pasa al segundo elemento y busca el mínimo en la subtabla que va desde el elemento siguiente hasta el final de la tabla. Ese mínimo lo intercambia por el segundo elemento (si es menor). Así sucesivamente hasta que la subtabla no tiene elementos. Nótese que en cada interacción del bucle (sea i el parámetro del for), entonces vamos sacando los i primeros mínimos de la tabla total.

En resumidas cuentas, SelectSort funciona porque se encarga de ir almacenando los primeros mínimos hasta que no quedan (primero almacena el primero, luego el segundo, luego el tercero y así sucesivamente).

6.3 ¿Por qué el bucle exterior de SelectSort no actúa sobre el último elemento de la tabla?

Porque la subtabla en la que hay que buscar el mínimo tiene 0 elementos. Una vez llegas al último elemento de la tabla, ya no hay más elementos y todos los anteriores a él son los n-1 números menores que el último elemento de la tabla. Por tanto, la tabla ya está ordenada para cuando se llega al elemento n.

6.4 ¿Cuál es la operación básica de SelectSort?

La operación básica del SelectSort es, tal y como se ha expuesto en el apartado 3, es la CDC que compara si el elemento seleccionado como el mínimo hasta el momento, es menor que el siguiente elemento en el que buscamos. En el pseudocódigo aportado en teoría: "Si T[j]<T[min]". Es claramente la operación que más veces se ejecuta.

6.5 Dar tiempos de ejecución en función del tamaño de entrada n para el caso peor WSS (n) y el caso mejor BSS (n) de SelectSort.

En base a lo visto en teoría y a una mera observación del pseudocódigo, SelectSort realiza la OB el mismo número de veces independientemente de cómo de desordenado u ordenando este el array. Es decir, el caso peor, caso mejor y caso medio tienen exactamente el mismo tae.

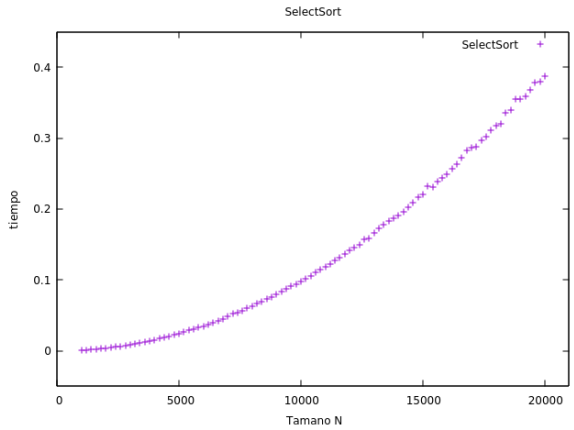
La razón es que el algoritmo siempre busca el mínimo en la subtabla correspondiente y, la única manera de hacerlo, es recorriendo toda la subtabla. Como esto va a pasar independientemente de la tabla en sí, el número de veces solo depende del tamaño, el tae para el caso peor, medio y mejor para un mismo tamaño son iguales.

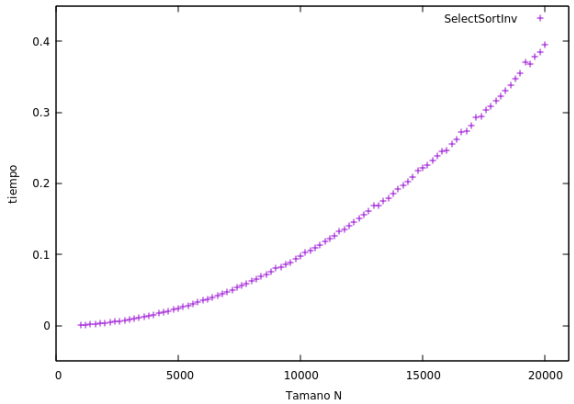
$$W_{SS}(N) = B_{SS}(N) = A_{SS}(N) = \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} 1 = \sum_{i=1}^{N-1} N - i = \sum_{i=1}^{N-1} N - \sum_{i=1}^{N-1} i =$$

$$= N^2 - \frac{N(N-1)}{2} = \frac{N^2}{2} - \frac{N}{2} = \frac{N^2}{2} + O(N)$$

6.6 Compara los tiempos obtenidos para SelectSort y SelectSortInv, justifica las similitudes o diferencias entre ambos (es decir, indicad si las gráficas son iguales o distintas y por qué)

Graficas SelectSort y SelectSortInv





Tras programar y ejecutar SelectSortInv, que utiliza el mismo algoritmo que SelectSort pero ordenando la permutación de mayor a menor, se han realizado dos gráficas comparativas del tiempo de ejecución de SelectSort y SelectSortInv respectivamente. Como se puede apreciar, ambas son cuadráticas, y el tiempo de ejecución para una misma entrada coincide para ambos algoritmos, luego ambos son exactamente igual de eficientes. Esto es debido a que el algoritmo es en definitiva el mismo, pero empezando por el final de la permutación.

Por tanto, se puede deducir que SS ~ SSInv, pues SS=O(SSInv) y SSInv=O(SS) y, además,

$$\lim_{x \to -\infty} \frac{SS}{SSInv} = 1$$

7. Conclusiones finales.

Tras la realización de la practica 1 de análisis de algoritmos, hemos adquirido nuevos conocimientos respecto a la eficiencia y la ejecución de los algoritmos de ordenación local, en particular de SelectSort y SelectSortInv. Además, hemos profundizado en el uso de las diversas herramienta que teníamos a nuestra disposición, como gnu plot, herramienta grafica para la representación de datos; o gdb, para la depuración de código.

En cuanto al cálculo de números aleatorios, hemos recurrido al libro *Numerical Recipies in C*, del cual hemos aprendido diversas formas de obtener *random numbers* y las ventajas y desventajas del uso de rand(), además de la equipotencia de dichos métodos.

Con respecto a la obtención de datos para las simulaciones, la metodología aplicada y el uso de estructuras para guardar dichos datos, nos han enseñado un nuevo enfoque de programación, aplicable para próximos proyectos.

Finalmente, la división del trabajo entre los participantes, incentiva una colaboración y un compromiso mutuo que nos prepara para poder participar en futuros proyectos académicos.