Análisis de Algoritmos 2022/2023

Práctica 2

Javier Jiménez y Pablo Fernández, Grupo 1282

Código	Gráficas	Memoria	Total

1. Introducción

En esta práctica analizamos dos algoritmos (y algunas variaciones) no locales de ordenación, comparando sus rendimientos en diferentes casos para identificar las ventajas y desventajas de utilizar uno u otro.

2. Objetivos

El objetivo de la práctica es comparar el rendimiento de los algoritmos de ordenación MergeSort y QuickSort, teniendo en cuenta para este último tres estrategias distintas para la elección del pivote. Primero habrá que implementar en C estos algoritmos y comprobar su buen funcionamiento. Posteriormente, analizaremos los casos mejor, peor y medio de todos ellos en tiempo de ejecución abstracto y puro.

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 13 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.

3.1 Implementación de MergeSort

Programamos una función recursiva `MergeSort` que implementa directamente el algoritmo estudiado en la clase de teoría, con su función auxiliar de combinación `merge`. Para la cuantificación del rendimiento vamos contando las operaciones básicas (CDCs) realizadas desde la base de la recursión hacia arriba.

Comprobamos el buen funcionamiento de esta función con el programa `test_sort.c`, que no es más que con una versión del programa `exercise4.c` modificada para que pueda probar cualquier algoritmo de ordenación que se le indique de entre los que hemos implementado hasta ahora (SelectSort, SelectSortInv, MergeSort, QuickSort_v1, QuickSort_v2 o QuickSort_v3).

3.2 Implementación de QuickSort y variaciones

Programamos una función genérica recursiva `_QuickSort` que implementa directamente el algoritmo estudiado en la clase de teoría, tomando como argumento la función `median` de elección de pivote. Para la cuantificación del rendimiento vamos contando las operaciones básicas (CDCs) realizadas desde la base de la recursión hacia arriba.

Esta función `_QuickSort` utiliza una función auxiliar `partition` que es la que en última instancia recibe la función `median` correspondiente y modifica la funcionalidad, pues el resto de la funcionalidad es común a las tres versiones de QuickSort que vamos a implementar.

Para las distintas versiones de `QuickSort` que queremos probar, programamos unas funciones de tipo wrapper que simplemente llaman a la función genérica `_QuickSort` pasándole la función de elección de pivote correspondiente. La nomenclatura que hemos adoptado es la siguiente:

1. `QuickSort_v1` utiliza la función `median` básica (el pivote es el primer elemento).
2. `QuickSort_v2` utiliza la función `median_avg` (el pivote es el elemento central).
3. `QuickSort_v3` utiliza la función `median_stat` (el pivote es la mediana de los elementos primero, último y central).

Las llamadas a las dos primeras funciones (`median` y `median_avg`) no aportan nada al coste abstracto del algoritmo, pues no realizan ninguna comparación de claves, así que devuelven el valor 0 si no hay ningún error. Sin embargo, la función `median_stat` sí que tiene un coste de entre 2 y 3 CDCs que es su valor de retorno y se debe tener en cuenta.

De nuevo, comprobamos el buen funcionamiento de las tres versiones utilizando el programa `test_sort.c`.

3.3 Rendimiento sobre permutaciones aleatorias de MergeSort y QuickSort

Para ser estrictos en la comparación del rendimiento, creamos la función `generate_sorting_times_n`, una versión modificada de la función `generate_sorting_times` que toma n algoritmos de ordenación y ejecuta las pruebas de rendimiento que ya diseñamos en la práctica 1 pero utilizando las mismas permutaciones para todos los algoritmos. De esta manera, los datos

de rendimiento que comparamos en las gráficas de varios algoritmos son los obtenidos exactamente con las mismas entradas aleatorias.

Ha sido necesaria una función auxiliar `copy_permutations` que crea una copia de un conjunto de permutaciones y para esta a su vez otra `copy_perm` que crea una copia de una única permutación.

También una función `average_sorting_time_alt` que funciona exactamente igual que `average_sorting_time` pero en vez de generar las permutaciones en su interior, las recibe como argumento. Esta función también la usaremos para los casos peores.

Esta función `generate_sorting_times_n` se llama desde el programa `stats_all.c`, que le pasa las funciones `MergeSort`, `QuickSort_v1`, `QuickSort_v2` y `QuickSort_v3`. Ejecutamos las pruebas de rendimiento para distintos intervalos, incrementos entre tamaños y cantidades de permutaciones. Los datos obtenidos los representamos en las gráficas del apartado 5.

3.4 Caso peor de MergeSort

Para analizar el caso peor del algoritmo MergeSort, implementamos la función `generate_mergesort_worst_perm`, que genera de forma recursiva la permutación que nos da el peor rendimiento de MergeSort para un tamaño $N = 2^k$ (generar esto recursivamente es relativamente sencillo, a diferencia de tamaños arbitrarios). Lo que hemos buscado al diseñar esta función es que en cada llamada de `MergeSort` a la función `merge`, tenga que realizar todas las CDCs posibles; en otras palabras, que las subtablas a combinar tengan el orden de los elementos alternado, como por ejemplo es el caso por ejemplo con $T_1 = [1, 3, 5, 7]$ y $T_2 = [2, 4, 6, 8]$.

Para las pruebas hemos creado una función específica `generate_sorting_times_mergesort_worst_perm`, que recibe como argumento una función de ordenación y dos exponentes naturales $k_1 < k_2$. Se realizan sobre el algoritmo los tests usuales para los tamaños $N = 2^i$ con $i = k_1, \dots, k_2$, pero utilizando en este caso la permutación generada por `generate_mergesort_worst_perm` en lugar de permutaciones aleatorias. Esto lo hacemos con una función `generate_permutations_alt` que genera el número de permutaciones que queramos usando la función de generación que le pasemos, que en este caso será `generate_mergesort_worst_perm`, en lugar de `generate_perm`. Realizar las pruebas del caso peor con más de una permutación es útil para suavizar la gráfica del tiempo puro de ejecución al hacer la media, aunque el número de OBs sea el mismo siempre.

Esta función se llama desde el programa `stats_mergesort_worst.c`, que nos permite elegir el algoritmo que queremos pasarle. Lo ejecutamos para los cuatro algoritmos con potencias de $k_1 = 1$ hasta $k_2 = 20$. Los datos obtenidos los representamos en las gráficas del apartado 5.

3.5 Caso peor de QuickSort

Igual que con MergeSort, implementamos tres funciones:

1. `generate_QuickSort_worst_perm_v1`, que genera la permutación que da el peor caso para `QuickSort_v1` (para cualquier tamaño). Esta permutación es la que ya está ordenada: $T = [1, 2, \dots, N]$.
2. `generate_QuickSort_worst_perm_v2`, que genera la permutación que da el peor caso para `QuickSort_v2` (para cualquier tamaño).
3. `generate_QuickSort_worst_perm_v3`, que genera la permutación que da el caso peor para `QuickSort_v3` (para cualquier tamaño par).

Las permutaciones de los casos 2 y 3 tienen una estructura un poco más compleja, pero tras leer un paper y un par de artículos sobre el tema acabamos llegando a un hilo de StackOverflow donde encontramos la manera de forzarla (algunos ejemplos en el apartado 5).

Básicamente, la estrategia que hay que tomar para diseñar las funciones anteriores es la siguiente: en cada llamada recursiva, el algoritmo de ordenación se vea obligado a hacer la partición con una de las subtablas resultantes de tamaño máximo. En los dos primeros casos este tamaño máximo es $N - 1$ (que coja como pivote el menor o mayor elemento de la tabla), mientras que en el tercero es $N - 2$ (que coja como pivote el segundo menor o mayor elemento, ya que por diseño es imposible que coja el menor o el mayor).

De nuevo, para las pruebas hemos creado una función específica `generate_sorting_times_QuickSort_worst_perm`, que recibe como argumento una función de ordenación, una de las funciones anteriores y el resto de datos habituales. Así, en este caso se realizan los tests que ya conocemos para todos los tamaños arbitrarios que queramos, utilizando la permutación generada por la función `generate_QuickSort_worst_perm_vi` recibida en lugar de permutaciones aleatorias. De nuevo empleamos la función `generate_permutations_alt`.

Esta función se llama desde el programa `stats_QuickSort_worst.c`, que nos permite elegir el algoritmo que queremos pasarle. Lo ejecutamos para los cuatro algoritmos en el intervalo 1-10001 con incrementos de 100. Los datos obtenidos los representamos en las gráficas del apartado 5.

4. Código fuente

4.1 Implementación de MergeSort

Mergesort (sorting.c)

```
int MergeSort(int* tabla, int ip, int iu) {
  int medio, ob1, ob2, ob3;

  /* Comprueba parámetros */
```

```c
  if (!tabla || ip < 0 || ip > iu)
    return ERR;

  /* Caso base */
  if (ip == iu)
    return 0;

  /* Recursión */
  medio = (ip + iu) / 2;
  ob1 = MergeSort(tabla, ip, medio);
  if (ob1 == ERR)
    return ERR;

  ob2 = MergeSort(tabla, medio + 1, iu);
  if (ob2 == ERR)
    return ERR;

  ob3 = merge(tabla, ip, iu, medio);
  if (ob3 == ERR)
    return ERR;

  return ob1 + ob2 + ob3;
}
```

merge (sorting.c)

```c
int merge(int *tabla, int ip, int iu, int imedio) {
  int *aux, i, j, k, ob = 0;

  /* Comprobación de parámetros */
  if (!tabla || ip < 0 || ip > iu || imedio < ip || imedio > iu)
    return ERR;

  /* Reserva memoria para tabla auxiliar */
  aux = (int *) malloc((iu - ip + 1) * sizeof(int));
  if (!aux)
    return ERR;

  /* Combina las tablas */
  for (i = ip, j = imedio + 1, k = 0; i <= imedio && j <= iu; k++) {
    if (tabla[i] < tabla[j]){
      aux[k] = tabla[i];
      i++;
    } else {
      aux[k] = tabla[j];
      j++;
    }
    ob++;
  }
```

4.2 Implementación de QuickSort

Función privada _QuickSort (sorting.c)

```
int _QuickSort(int *tabla, int ip, int iu, pfunc_median median) {
  int medio, ob1, ob2 = 0, ob3 = 0;

  /* Comprueba parámetros */
  if (!tabla || !median || ip < 0 || ip > iu)
    return ERR;

  /* Caso base */
  if (ip == iu)
    return OK;

  /* Recursión */
  ob1 = partition(tabla, ip, iu, &medio, median);
  if (ob1 == ERR)
    return ERR;

  if (ip < medio - 1)
    ob2 = _QuickSort(tabla, ip, medio - 1, median);
  if (ob2 == ERR)
    return ERR;

  if (medio + 1 < iu)
    ob3 = _QuickSort(tabla, medio + 1, iu, median);
  if (ob3 == ERR)
    return ERR;

  return ob1 + ob2 + ob3;

}
```

QuickSort_v1/v2/v3 (sorting.c)

```
int QuickSort_v1(int *tabla, int ip, int iu){
  return _QuickSort(tabla, ip, iu, median);
}

int QuickSort_v2(int *tabla, int ip, int iu){
  return _QuickSort(tabla, ip, iu, median_avg);
}

int QuickSort_v3(int *tabla, int ip, int iu){
  return _QuickSort(tabla, ip, iu, median_stat);
}
```

partition (sorting.c)

```c
int partition(int* tabla, int ip, int iu, int *pos, pfunc_median
median_func) {
  int medio, i, k, ob = 0;

  /* Comprueba parámetros */
  if (!tabla || !median_func || ip < 0 || ip > iu || !pos)
    return ERR;

  if (median_func(tabla, ip, iu, &medio) == ERR)
    return ERR;

  k = tabla[medio];
  _swap(tabla + ip, tabla + medio);
  medio = ip;

  for (i = ip + 1; i <= iu; i++) {
    if (tabla[i] < k) {
      medio++;
      _swap(tabla + i, tabla + medio);
    }
    ob++;
  }

  _swap(tabla + ip, tabla + medio);

  *pos = medio;

  return ob;
}
```

median (sorting.c)

```c
int median(int *tabla, int ip, int iu, int *pos) {
  /* Comprueba parámetros */
  if (!tabla || ip < 0 || ip > iu || !pos)
    return ERR;

  *pos = ip;
  return 0;
}
```

median_avg (sorting.c)

```c
int median_avg(int *tabla, int ip, int iu, int *pos) {
  /* Comprueba parámetros */
  if (!tabla || ip < 0 || ip > iu || !pos)
    return ERR;
```

```
  *pos = (ip + iu) / 2;
  return 0;
}
```

median_stat (sorting.c)

```
int median_stat(int *tabla, int ip, int iu, int *pos) {
  int im = (ip + iu) / 2, min, a1;

  /* Comprueba parámetros */
  if (!tabla || ip < 0 || ip > iu || !pos)
    return ERR;

  if (tabla[ip] < tabla[im]) {
    min = ip;
    a1 = im;
  } else {
    min = im;
    a1 = ip;
  }

  if (tabla[a1] < tabla[iu]) {
    *pos = a1;
    return 2;
  }

  if (tabla[min] < tabla[iu])
    *pos = iu;
  else
    *pos = min;

  return 3;
}
```

Función privada _swap (sorting.c)

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

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

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

4.3 Rendimiento sobre permutaciones aleatorias de MergeSort y QuickSort

average_sorting_time_alt (times.c)

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

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

  /* 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_n (times.c)

```
short generate_sorting_times_n(pfunc_sort *method, char** file, int
num_func, int num_min, int num_max, int incr, int n_perms){
  PTIME_AA sorting_times;
  int i, j, k, flag, num_ptimes, **perms, **copy_perms;

  /* Comprueba parámetros */
  if (!method || !file || num_func < 1|| num_min <= 0 || num_max <
num_min || incr <= 0 || n_perms <= 0)
    return ERR;

  for (i = 0; i < num_func; i++)
    if (!method[i] || !file[i])
      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 * num_func *
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++){
    perms = generate_permutations(n_perms, i);
    if (!perms)
      flag = ERR;
    for (k = 0; k < num_func && flag == OK; k++){
      copy_perms = copy_permutations(perms, n_perms, i);
      if (!perms)
        flag = ERR;
      if (flag == OK)
        flag = average_sorting_time_alt(method[k], n_perms, i,
copy_perms, sorting_times + k * num_ptimes + j);
      free_perms(copy_perms, n_perms);
    }

    free_perms(perms, n_perms);
```

```
  }
```

copy_perm (permutations.c)

```
int *copy_perm(int *array, int N){
  int *copy, i;

  if (N <= 0 || !array)
    return NULL;

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

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

  return copy;
}
```

copy_permutations (permutations.c)

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

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

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

  /* Copia las permutaciones */
  for (i = 0; i < n_perms && flag == OK; i++){
    perms[i] = copy_perm(array[i], N);
    if (perms[i] == NULL) {
      flag = ERR;
    }
  }
  /* Comprueba errores en la copia de permutaciones */
  if (flag == ERR) {
    for (i -= 2; i >= 0; i--) {
      free(perms[i]);
```

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

  return perms;

}
```

Función privada free_perms

```
void free_perms(int **array, int num){
  int i;

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

  for (i = 0; i < num; i++)
    free(array[i]);

  free(array);
}
```

4.4 Caso peor de MergeSort

generate_sorting_times_mergesort_worst_perm (times.c)

```
short generate_sorting_times_mergesort_worst_perm(pfunc_sort method,
char* file, int pot_min, int pot_max, int n_perms){
  PTIME_AA sorting_times;
  int i, j, flag, num_ptimes , **perm;

  /* Comprueba parámetros */
  if (!method || !file || pot_min < 0 || pot_max < pot_min || n_perms <
1)
    return ERR;

  /* Reserva de memoria para las estructuras que almacenan los datos */
  num_ptimes = pot_max - pot_min + 1;
  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 = pot_min, j = 0, flag = OK; i <= pot_max && flag == OK; i++,
j++){
    perm = generate_permutations_alt(generate_mergesort_worst_perm,
n_perms, i);
    if (!perm)
```

```
      flag = ERR;
    if (flag == OK)
      flag = average_sorting_time_alt(method, n_perms, pow(2, i), perm,
sorting_times + j);
    free_perms(perm, n_perms);
  }

  /* 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;
}
```

generate_permutations_alt (permutations.c)

```
int ** generate_permutations_alt(pfunc_perm func_perm, int n_perms, int
N) {
  int **perms, i, flag = OK;

  /* Comprueba parámetros */
  if (!func_perm || 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] = func_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]);
      return NULL;
    }
    free(perms);
```

```
  }

  return perms;
}
```

Función privada _generate_mergesort_worst_perm_rec (permutations.c)

```
void _generate_mergesort_worst_perm_rec(int pot, int* array) {
  int i, size;

  /* Caso base */
  if (pot == 0) {
    array[0] = 1;
    return;
  }

  /* Recursión */
  size = pow(2, pot - 1);
  _generate_mergesort_worst_perm_rec(pot - 1, array);
  _generate_mergesort_worst_perm_rec(pot - 1, array + size);

  for (i = 0; i < size; i++) {
    array[i] = 2 * array[i] - 1;
    array[size + i] = 2 * array[size + i];
  }
}
```

generate_mergesort_worst_perm (permutations.c)

```
int *generate_mergesort_worst_perm(int pot) {
  int *array, size = (int) pow(2, pot);

  /* Comprueba parámetros */
  if (pot < 0)
    return NULL;

  /* Reserva memoria para la permutación */
  array = (int *) malloc(size * sizeof(int));
  if (!array)
    return NULL;

  /* Genera la función recursivamente */
  _generate_mergesort_worst_perm_rec(pot, array);

  return array;
}
```

4.5 Caso peor de QuickSort

generate_sorting_times_QuickSort_worst_perm (times.c)

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

  /* Comprueba parámetros */
  if (!method || !worst_perm || !file || num_min <= 0 || num_max <
num_min || incr <= 0)
    return ERR;
  if (worst_perm == generate_QuickSort_worst_perm_v3 && (num_min % 2 ||
incr % 2))
    return ERR;

  /* Cálculo de los sorting times para cada tamaño */
  /* 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++){
    perm = generate_permutations_alt(worst_perm, n_perms, i);
    if (!perm)
      flag = ERR;
    if (flag == OK)
      flag = average_sorting_time_alt(method, n_perms, i, perm,
sorting_times + j);
    free_perms(perm, n_perms);
  }


  /* 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;
}
```

generate_QuickSort_worst_perm_v1 (permutations.c)

```
int *generate_QuickSort_worst_perm_v1(int N){
  int *array, i;

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

  /* Reserva memoria para la permutación */
  array = (int*) malloc(N * sizeof(int));
  if (!array)
    return NULL;

  /* Genera la permutación */
  for (i = 0; i < N; i++){
    array[i] = i;
  }

  return array;
}
```

generate_QuickSort_worst_perm_v2 (permutations.c)

```
int *generate_QuickSort_worst_perm_v2(int N){
  int *p, *v, i, pivot;

  p = (int*) malloc(N * sizeof(int));
  if (!p)
    return NULL;

  v = (int*) malloc(N * sizeof(int));
  if (!v){
    free(p);
    return NULL;
  }

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

  for (i = 0; i < N; i++){
    pivot = (N - 1 + i) / 2;
    v[p[pivot]] = i;
    __swap(p + pivot, p + i);
  }
```

```
  free(p);
  return v;
}
```

generate_QuickSort_worst_perm_v3 (permutations.c)

```
int *generate_QuickSort_worst_perm_v3(int N){
  int *p, *v, i, pivot0, pivot1;

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

  /* Reserva memoria para las permutaciones y el índice de referencia */
  p = (int *) malloc(N * sizeof(int));
  if (!p)
    return NULL;

  v = (int *) malloc(N * sizeof(int));
  if (!v){
    free(p);
    return NULL;
  }

  /* Índice de referencia de los números */
  for (i = 0; i < N; i++)
    p[i] = i;

  /* Generación de la permutación de 0 a N-1 */
  for(i = 0; i < N; i += 2){
    pivot0 = i;
    pivot1 = (i + N - 1)/2;
    v[p[pivot1]] = i + 1;
    v[p[pivot0]] = i;
    __swap(p + pivot1, p + i + 1);
  }

  if(i == N){
    v[N -1] = i - 1;
  }

  /* Incremento de una unidad para tener la permutación de 1 a N */
  for (i = 0; i < N; i++)
    v[i]++;

  free(p);

  return v;
}
```

5. Resultados, Gráficas

5.1 Implementación de MergeSort

Mergesort funciona correctamente, sin ninguna fuga de memoria y ordenando correctamente la permutación:

```
==6253== Memcheck, a memory error detector
==6253== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6253== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==6253== Command: ./test_sort -size 10 -func 2
==6253==
Practice number 2, Sorting test
Done by: Javier Jiménez, Pablo Fernández
Group: 1202
1       2       3       4       5       6       7       8       9       10
==6253==
==6253== HEAP SUMMARY:
==6253==     in use at exit: 0 bytes in 0 blocks
==6253==   total heap usage: 11 allocs, 11 frees, 1,200 bytes allocated
==6253==
==6253== All heap blocks were freed -- no leaks are possible
==6253==
==6253== For lists of detected and suppressed errors, rerun with: -s
==6253== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

5.2 Implementación de QuickSort

QuickSort funciona correctamente, ordenando correctamente la permutación, independientemente del pivote elegido.

1. QuickSort_v1 (pivote el primer elemento):

```
==6306== Memcheck, a memory error detector
==6306== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6306== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright
info
==6306== Command: ./test_sort -size 10 -func 3
==6306==
Practice number 2, Sorting test
Done by: Javier Jiménez, Pablo Fernández
Group: 1202
1       2       3       4       5       6       7       8       9
10
==6306==
==6306== HEAP SUMMARY:
==6306==     in use at exit: 0 bytes in 0 blocks
==6306==   total heap usage: 2 allocs, 2 frees, 1,064 bytes allocated
==6306==
==6306== All heap blocks were freed -- no leaks are possible
==6306==
==6306== For lists of detected and suppressed errors, rerun with: -s
==6306== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

2. QuickSort_v2 (pivote el elemento central):

```
==6317== Memcheck, a memory error detector
==6317== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
```

```
==6317== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright
info
==6317== Command: ./test_sort -size 10 -func 4
==6317==
Practice number 2, Sorting test
Done by: Javier Jiménez, Pablo Fernández
Group: 1202
1       2       3       4       5       6       7       8       9
10
==6317==
==6317== HEAP SUMMARY:
==6317==     in use at exit: 0 bytes in 0 blocks
==6317==   total heap usage: 2 allocs, 2 frees, 1,064 bytes allocated
==6317==
==6317== All heap blocks were freed -- no leaks are possible
==6317==
==6317== For lists of detected and suppressed errors, rerun with: -s
==6317== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

3. QuickSort_v3 (pivote la mediana de los elementos primero, central y último):

```
==6326== Memcheck, a memory error detector
==6326== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6326== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright
info
==6326== Command: ./test_sort -size 10 -func 5
==6326==
Practice number 2, Sorting test
Done by: Javier Jiménez, Pablo Fernández
Group: 1202
1       2       3       4       5       6       7       8       9
10
==6326==
==6326== HEAP SUMMARY:
==6326==     in use at exit: 0 bytes in 0 blocks
==6326==   total heap usage: 2 allocs, 2 frees, 1,064 bytes allocated
==6326==
==6326== All heap blocks were freed -- no leaks are possible
==6326==
==6326== For lists of detected and suppressed errors, rerun with: -s
==6326== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

5.3 Rendimiento sobre permutaciones aleatorias de MergeSort y QuickSort

Para las gráficas que vamos a exponer a partir de aquí, hemos utilizado una función de gnuplot que calcula la constante k con la que mejor se aproxima una función $f(x) = k \cdot g(x)$ a los datos graficados. Así, en los títulos de las gráficas indicamos la función a la que se asemejan los rendimientos de los algoritmos.

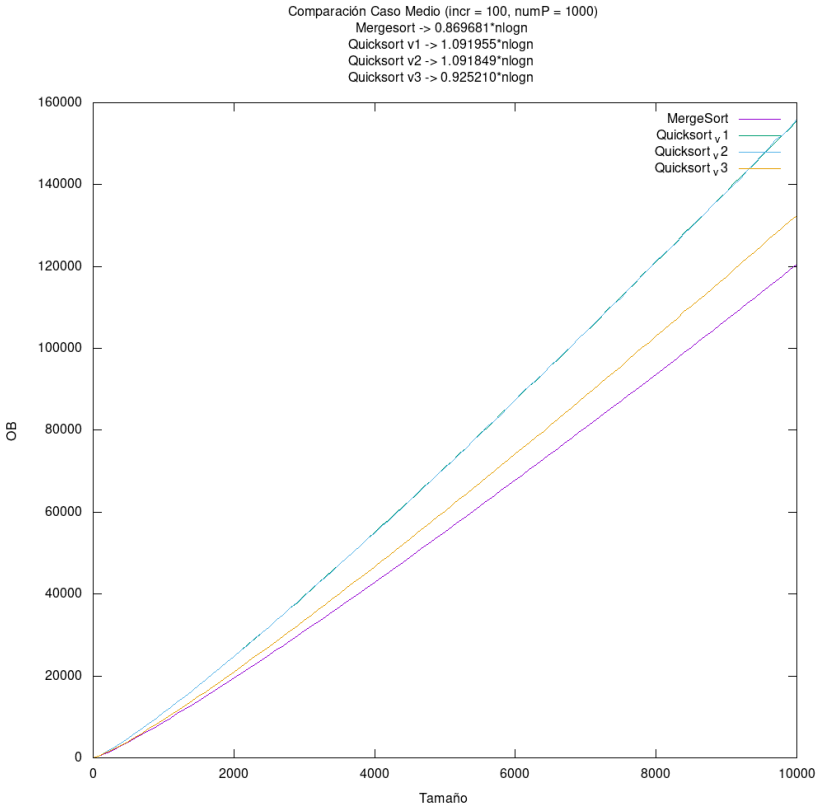
5.3.1 Rendimiento abstracto en el caso medio

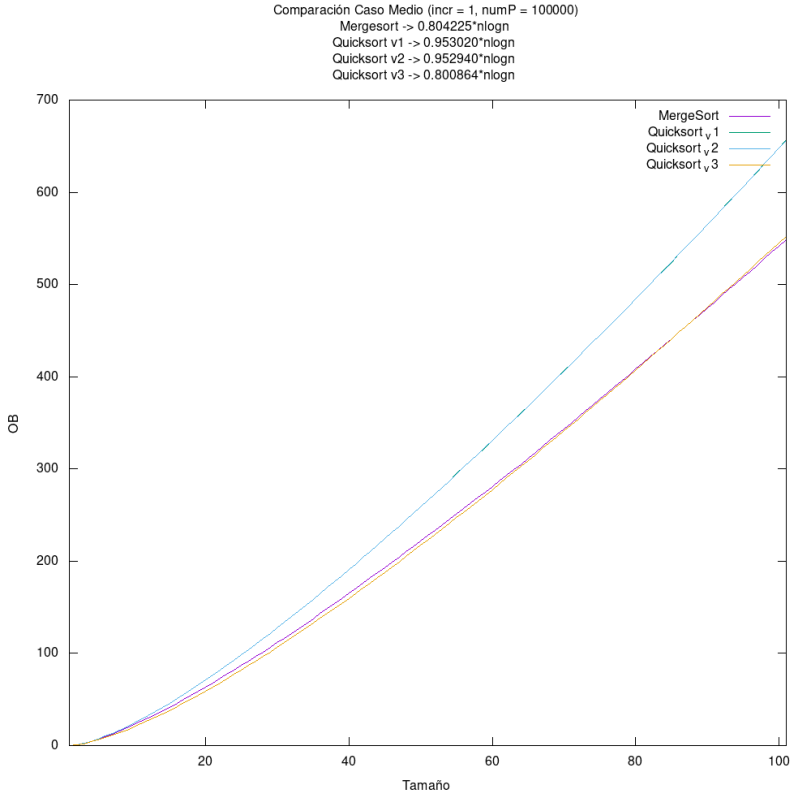
Sabemos por el análisis teórico que el caso medio de MergeSort y QuickSort son ambos loglineales:

$$A_{MS} = \Theta(N \log N)$$

$$A_{QS} = \Theta(N \log N)$$

Comparamos el promedio de operaciones básicas realizadas por cada algoritmo sobre permutaciones aleatorias.





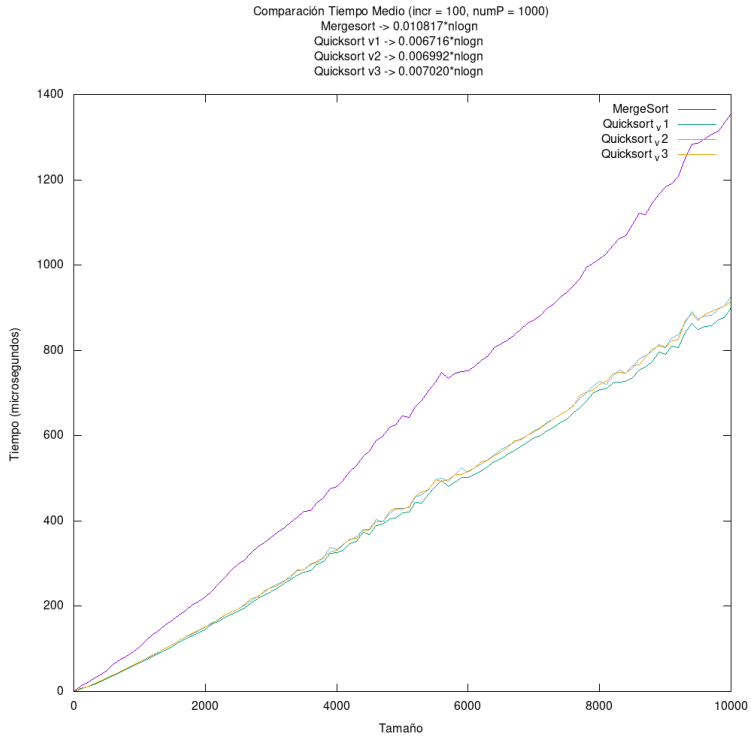
Observamos que efectivamente el rendimiento abstracto medio de todas las funciones es comparable a $N \log N$. Mergesort es el algoritmo más eficiente en cuanto a operaciones básicas, seguido del QuickSort_v3 (mediana de tres), y en último lugar tenemos el QuickSort_v1 y el QuickSort_v2 con un rendimiento prácticamente idéntico.

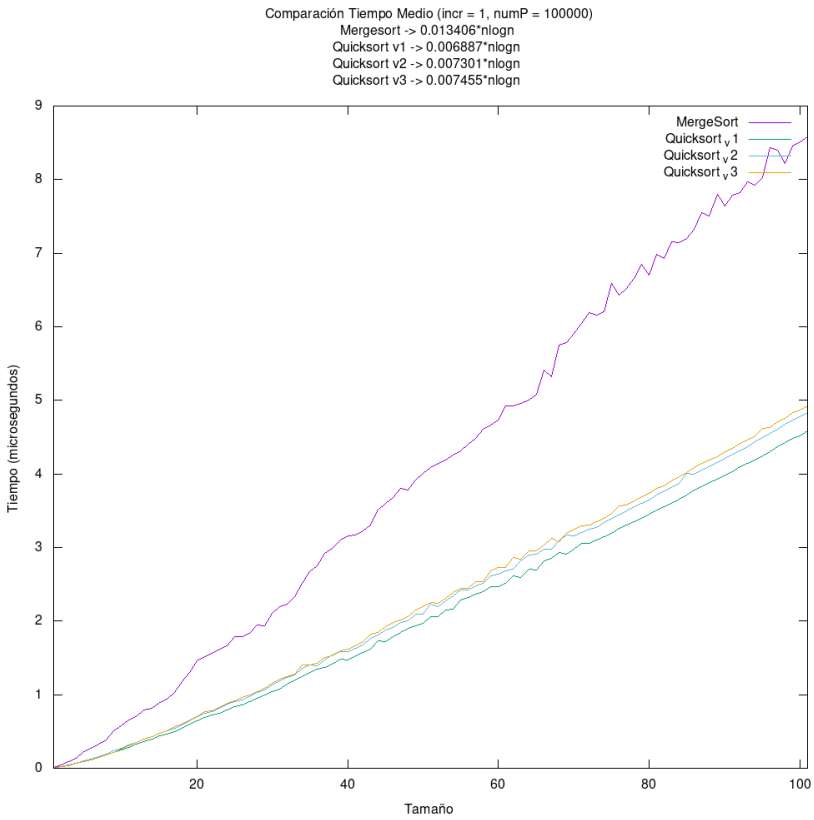
Si tomamos las operaciones básicas del Mergesort como referencia, el QuickSort_v1 y v2 son un 25% más costosos en permutaciones grandes, y un 19% en las pequeñas. En cambio, el QuickSort_v3 es un 6% más costoso sobre permutaciones grandes y tiene el mismo rendimiento en las pequeñas.

En la primera gráfica están representados tamaños desde el 1 hasta el 10001, y el Mergesort tiene una clara ventaja frente al QuickSort_v3 en cuanto a OBs, con un 6% de diferencia. Sin embargo, en la segunda gráfica con tamaños mucho más pequeños, del 1 al 101, la diferencia entre los dos es casi indistinguible.

Es decir, para tamaños pequeños parece teóricamente mejor utilizar el QuickSort_v3 ya que no necesita memoria adicional y el Mergesort sí. Sin embargo, para tamaños grandes, Mergesort ofrece una mejora en OBs a tener en cuenta, por lo que, si contamos con los suficientes recursos de memoria, podría ser interesante su uso. A continuación, veremos si este análisis abstracto se traduce en el rendimiento real.

5.3.2 Rendimiento puro en el caso medio





En estas gráficas podemos observar claramente el impacto que tiene en el rendimiento la reserva continua de memoria que realiza MergeSort. Aunque en abstracto pueda ser el algoritmo más eficiente porque realiza menos comparaciones de clave, lo cierto es que estar gestionando la memoria auxiliar y copiar los datos de un sitio a otro consume un tiempo muy considerable (entre un 61% y un 85% más) que QuickSort, en cualquier versión, ahorra por ser *in place*.

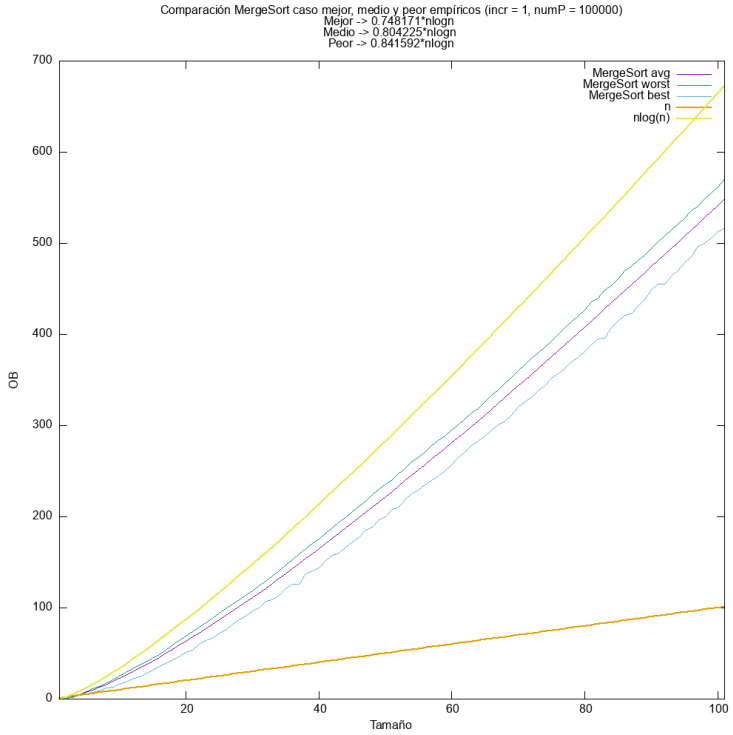
Entre las tres versiones de QuickSort no se aprecia una gran diferencia: las versiones 2 (pivote central) y 3 (pivote mediana de tres) van muy a la par, mientras que la versión 1 (pivote inicial) es ligeramente más rápido. En el apartado 6.2 comentamos un poco por qué puede ser.

Concluimos que, a pesar de lo que nos puede dar el análisis abstracto en términos de OBs, en la práctica es más eficiente utilizar QuickSort, ya que ahorramos tiempo y memoria.

5.3.4 Casos mejor, medio y peor empírico

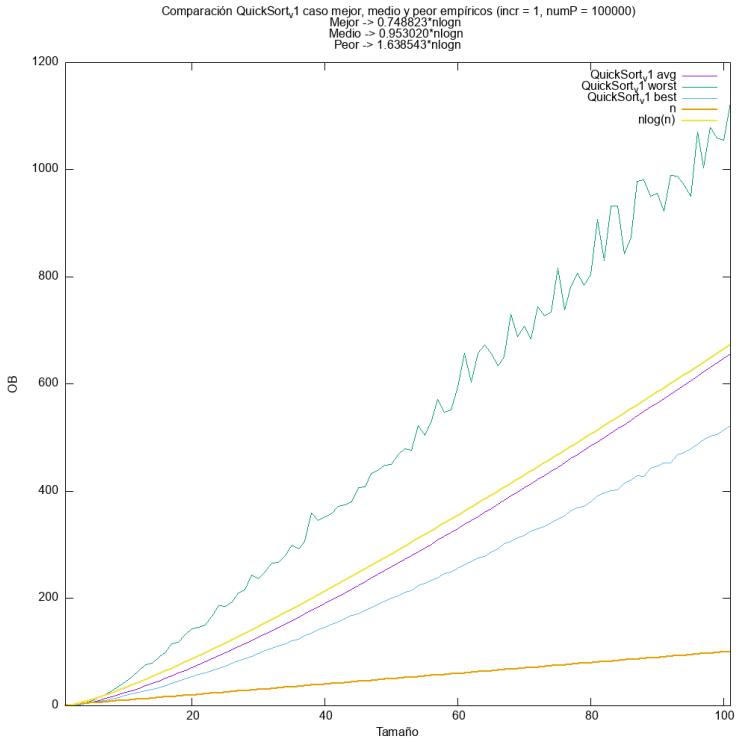
Para cada algoritmo, vamos a comparar el máximo, mínimo y el promedio de OBs realizadas al ejecutarlo sobre 100000 permutaciones aleatorias de tamaños de 1 a 101. Elegimos este intervalo porque los tamaños son lo bastante pequeños como para poder ordenar un número tan grande de permutaciones en un período de tiempo razonable. Si hiciéramos las pruebas con un número menor de permutaciones, disminuiría aún más la probabilidad de encontrar una permutación parecida al caso mejor o al peor.

Mergesort



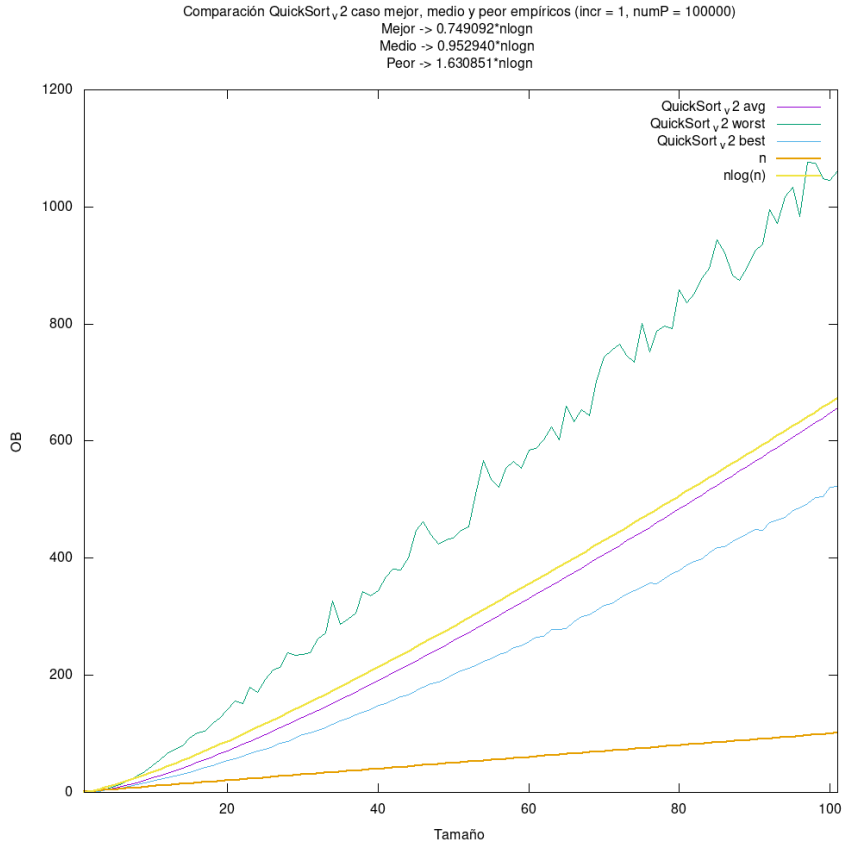
Observamos que en los tres casos tenemos un rendimiento comparable a $N \log N$, lo que concuerda con la teoría, ya que los casos mejor, peor y medio de MergeSort son de este orden. La desviación respecto al promedio de la peor permutación que ha salido es de un 5% mayor coste, mientras que la mejor es un 7% menor; una desviación relativamente pequeña.

QuickSort_v1



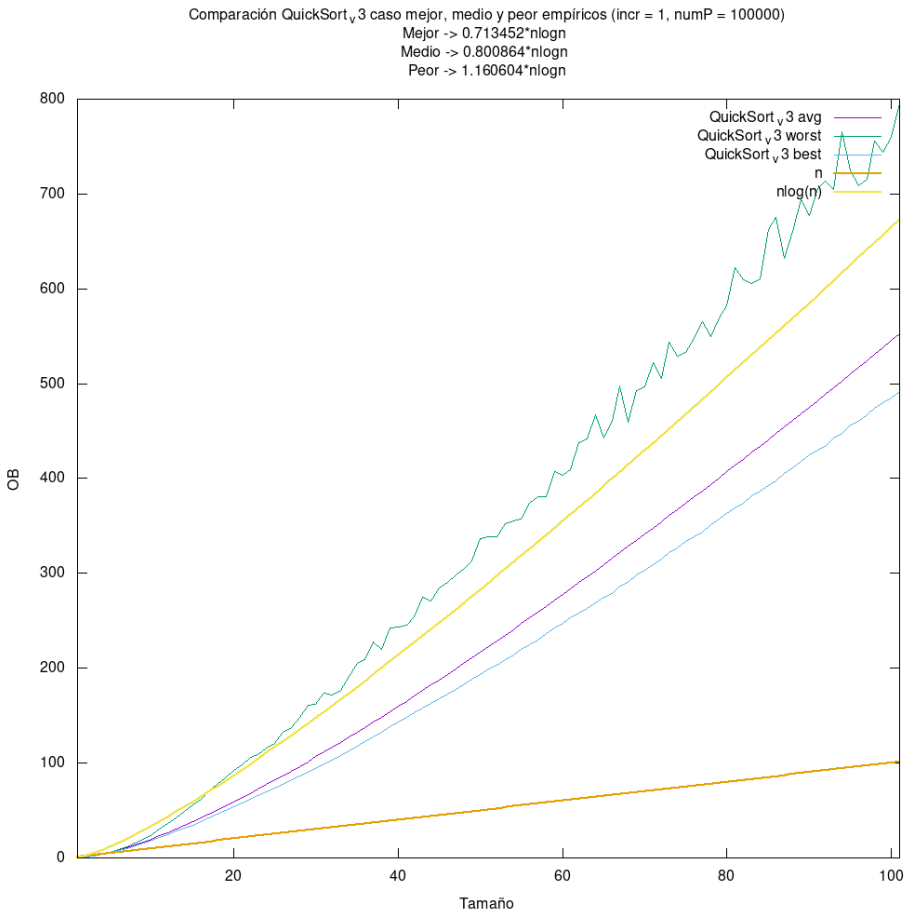
Como en MergeSort, en los tres casos el rendimiento es del orden de $N \log N$, pero aquí la desviación es bastante mayor: la peor permutación es un 72% más costosa que el promedio y la mejor permutación un 22% menos. Debemos hacer notar que para este algoritmo el caso peor real es cuadrático, pero empíricamente es muy complicado encontrarlo con permutaciones aleatorias, y por eso obtenemos un “caso peor” loglineal.

QuickSort_v2



El caso de QuickSort_v2 es prácticamente idéntico al QuickSort v1, lo que demuestra que independientemente de la posición que elijamos para el pivote, si esta es fija no apreciaremos mejora del rendimiento. Esto es así porque las permutaciones aleatorias son equiprobables, así que todos los elementos caen con la misma probabilidad en todas las posiciones.

QuickSort_v3

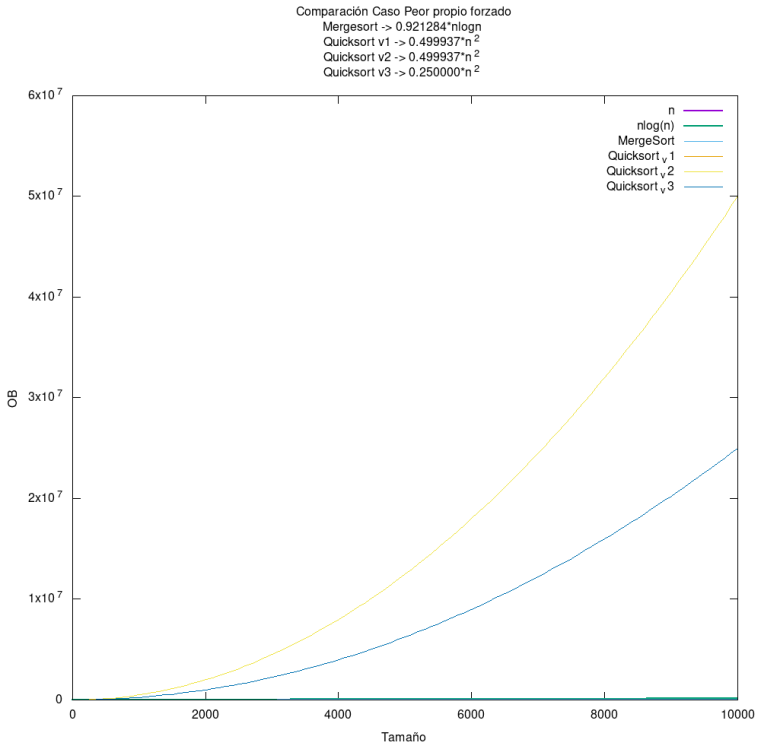


Para QuickSort_v3, observamos que su caso medio es un 16% más rápido que el de QuickSort_v1 y v2, por lo tanto, sí que hay cierta mejora en cuanto a operaciones básicas al elegir nuestro pivote comparando varios elementos. Si tomamos como referencia el caso medio del QuickSort_v3, la peor permutación que ha aparecido es un 45% más costosa, mientras que la mejor es un 11% menos. Podemos observar que el caso peor y mejor empíricos están más cerca del caso medio en el QuickSort_v3, lo que significa que tiene menos variabilidad que QuickSort_v1 y v2.

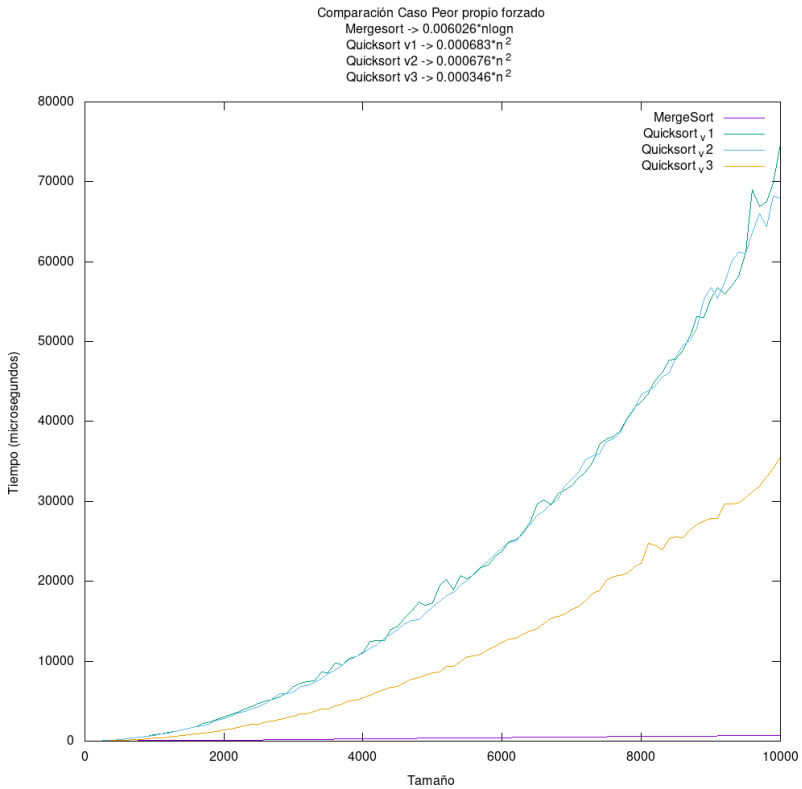
5.4 Caso peor teórico

Para forzar el caso peor teórico hemos utilizado las funciones descritas anteriormente que generan la peor permutación posible para el algoritmo correspondiente.

Las siguientes gráficas muestran el caso peor de cada función en operaciones básicas y tiempo.



Podemos observar que el caso peor en operaciones básicas del Mergesort es loglineal mientras que los QuickSort son cuadráticos. Los más costosos son el QuickSort_v1 y v2 que tardan el doble que el QuickSort_v3, esto es debido a que el último al hacer la comparación del primero, último y central, nunca va a coger el peor elemento como pivote, va a coger el segundo peor, por eso tiene la mitad de OBs.

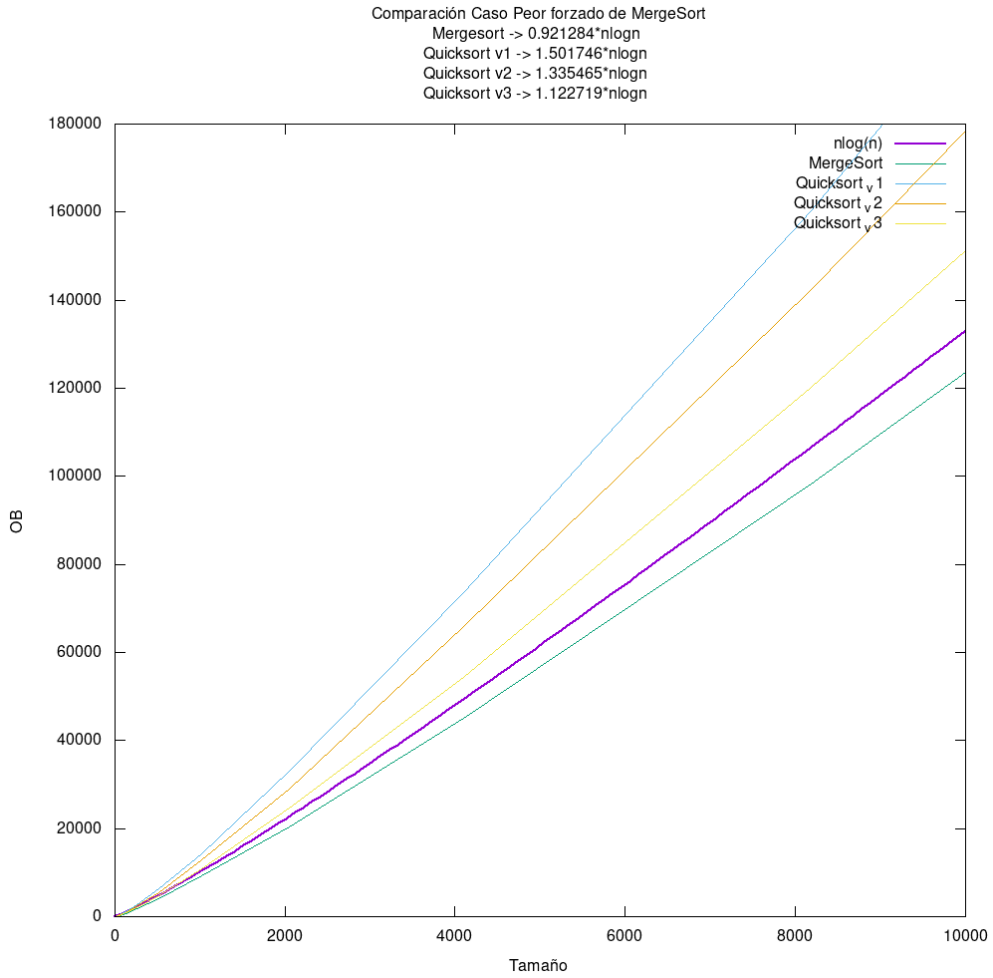


Podemos observar que la gráfica de tiempo se asemeja bastante a la gráfica de OBs, siendo el tiempo del Mergesort loglineal y el de los QuickSort cuadráticos. También se cumple que los QuickSort_v1 y v2 tardan el doble que el v3.

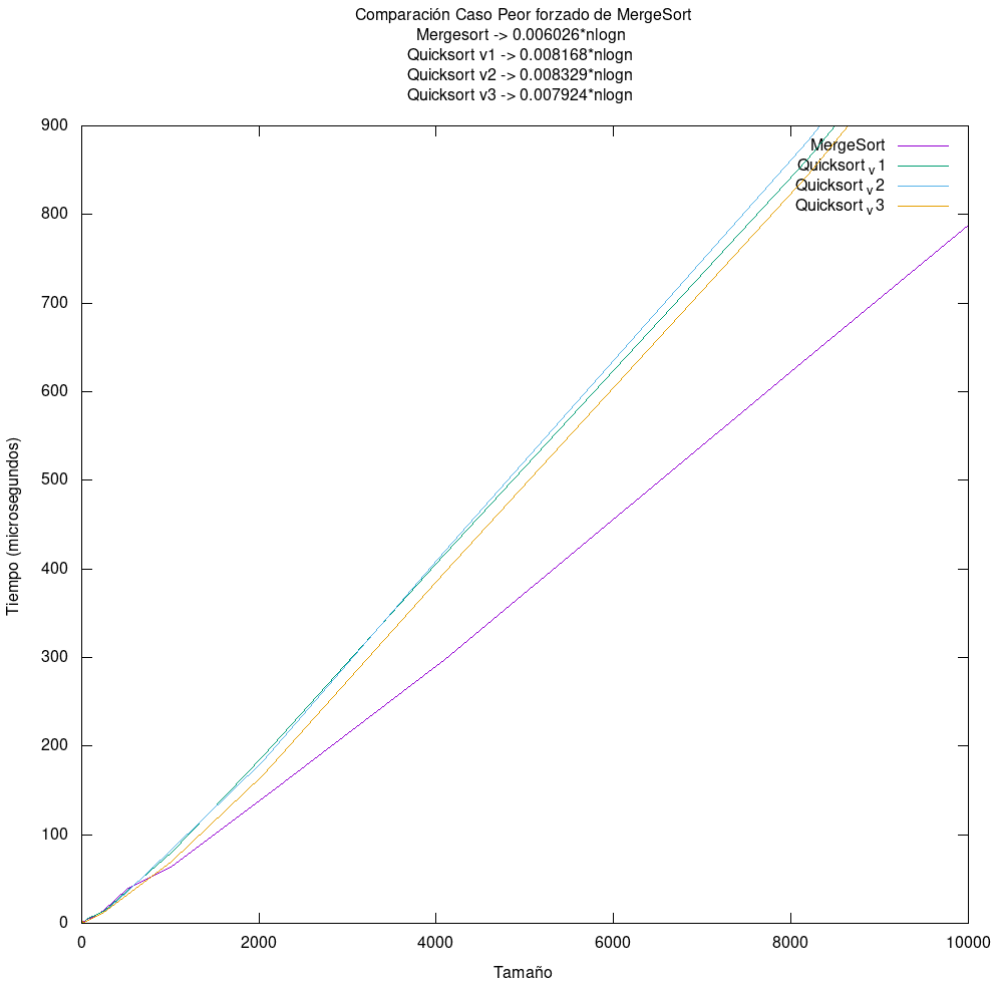
En conclusión, sale ampliamente victorioso el MergeSort, pero la relevancia de este análisis es relativa, ya que es poco probable que en un entorno de trabajo real nos encontremos con el caso peor de QuickSort de forma habitual.

5.5 Rendimiento con el peor caso teórico del Mergesort

En las siguientes gráficas está representado el rendimiento de Mergesort y QuickSort forzándolos con las peores permutaciones para el Mergesort. Aunque sea la misma permutación, los test están hechos con 50 permutaciones por tamaño para suavizar la gráfica de tiempo.



Sorprendentemente, aunque sea el peor caso del Mergesort, en cuanto a OBs, sigue siendo menos costoso que los QuickSort con una diferencia considerable son desde un 33% hasta un 63% más costoso. Seguramente esto se debe a que el peor caso del Mergesort es logarítmico mientras que el del QuickSort es cuadrático y este será un caso no muy bueno para QuickSort.

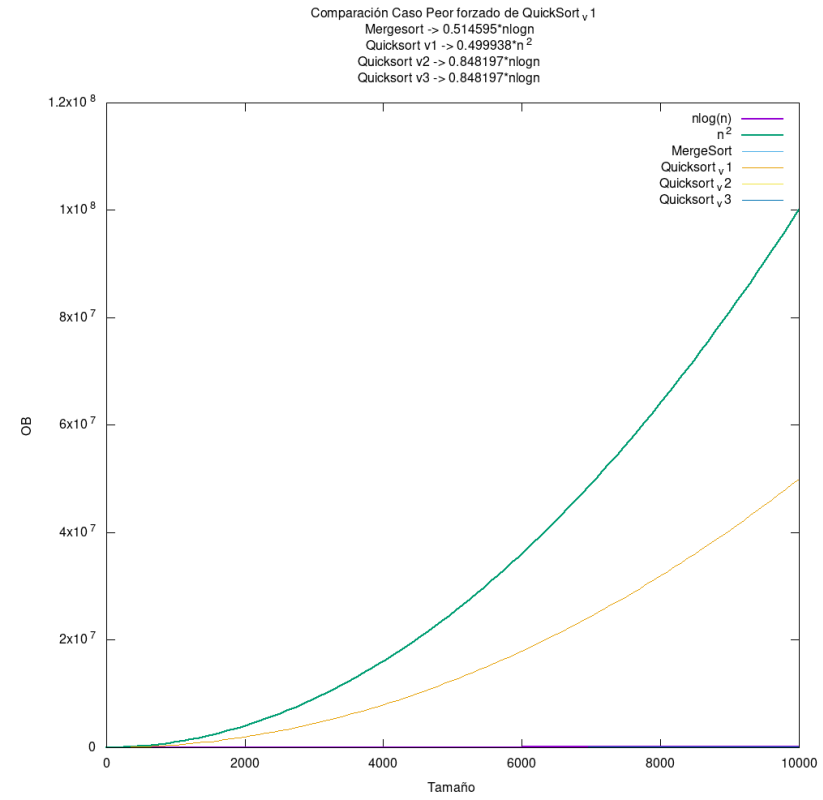


En cuanto al tiempo, el Mergesort tarda significativamente menos que los QuickSort, un 38% más aproximadamente. Esto demuestra que el Mergesort, aunque en el caso medio sea más lento en tiempo, con permutaciones poco favorables presenta un mejor desempeño que QuickSort. Podríamos decir que el rendimiento de MergeSort es “más estable".

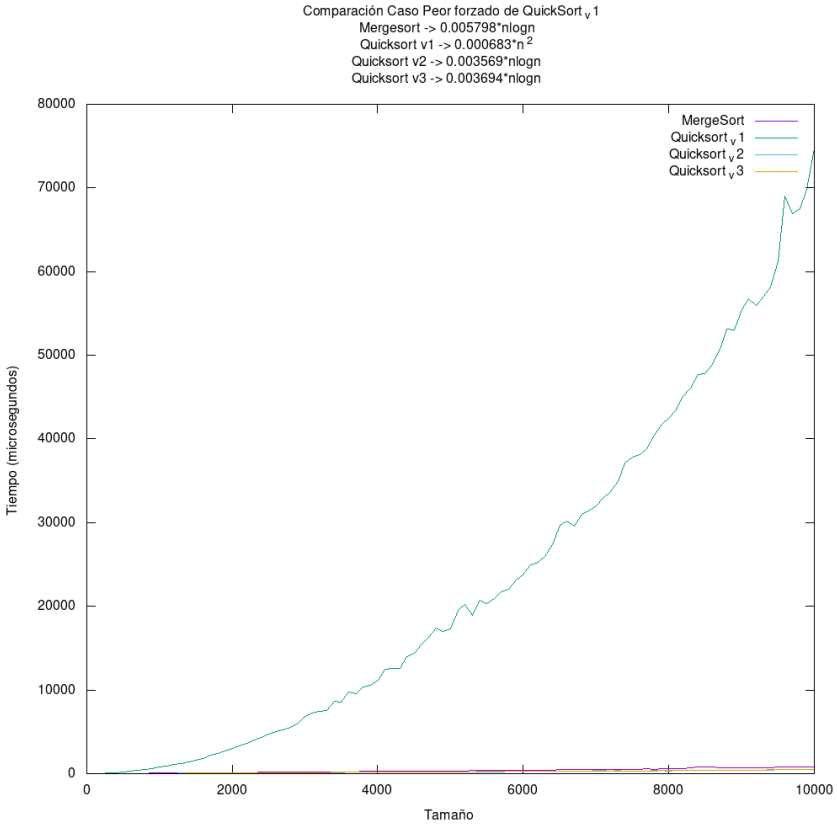
5.6 Rendimiento con el peor caso teórico del QuickSort

5.6.1 QuickSort_v1

En las siguientes gráficas está representado el rendimiento de Mergesort y QuickSort forzándolos con las peores permutaciones para el QuickSort_v1, es decir, con permutaciones ordenadas. Aunque sea la misma permutación, los test están hechos con 50 permutaciones por tamaño para suavizar la gráfica de tiempo.



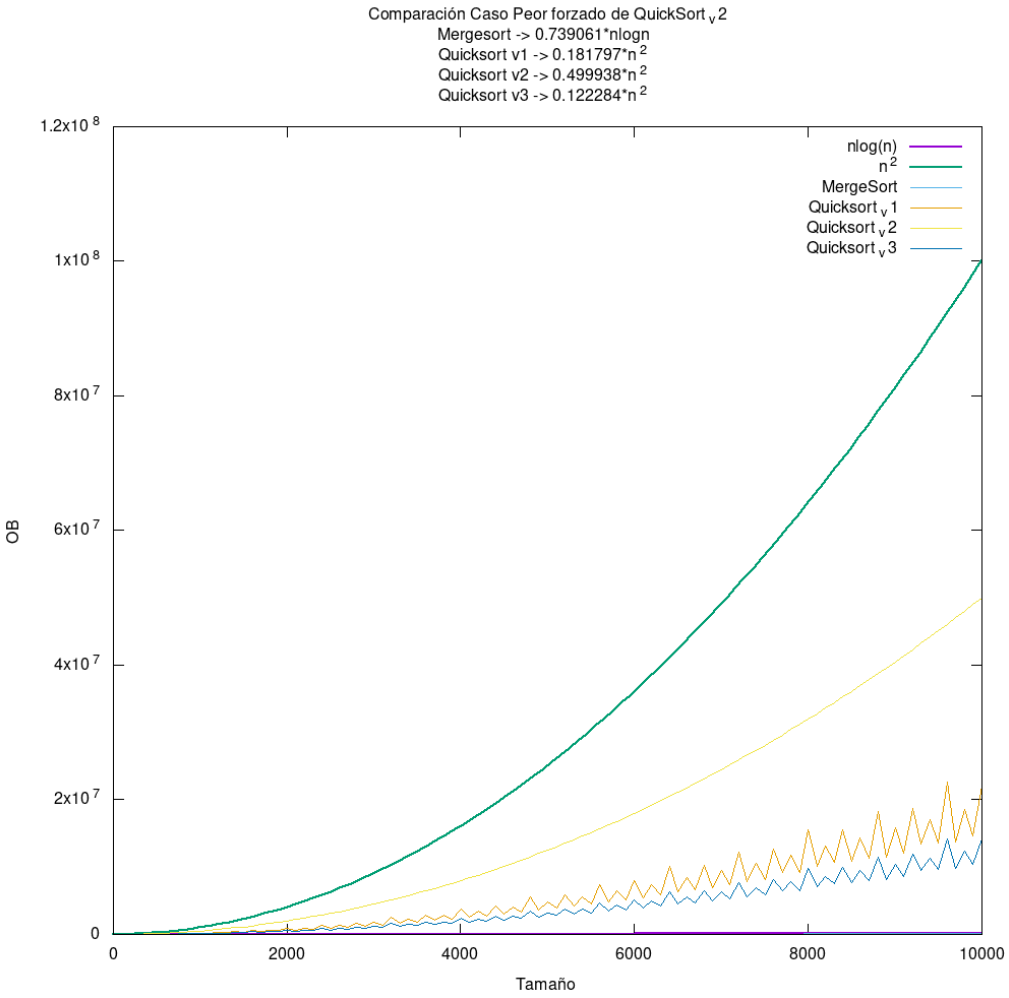
Podemos observar que las operaciones básicas de los QuickSort_v2 y v3 y Mergesort son logarítmicas, es decir son comparables con su rendimiento medio. Por lo tanto, esta permutación solo representa un problema para QuickSort_v1.



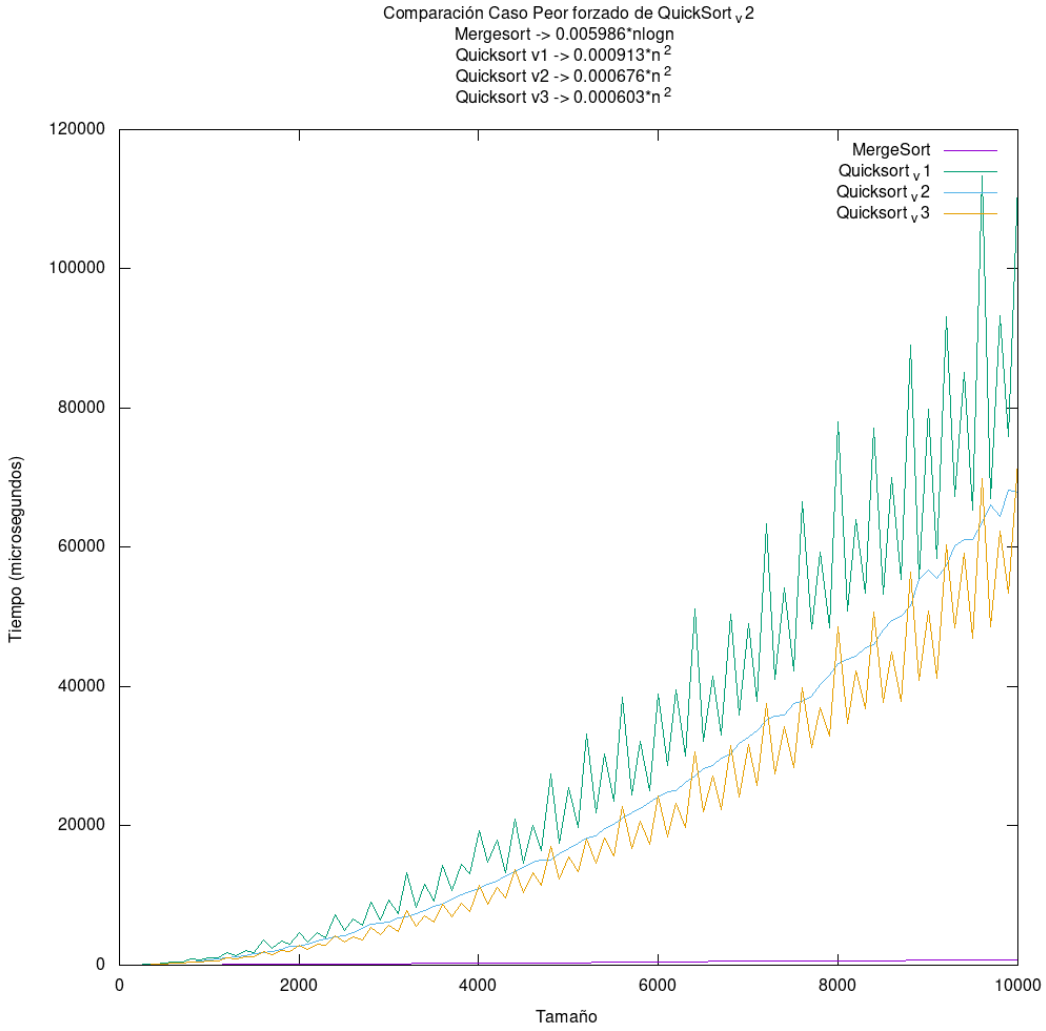
Al igual que con las OBs, el tiempo del QuickSort_v1 es cuadrático mientras que los del resto es loglineal y, por lo tanto, el tiempo que tardan es despreciable en comparación con el QuickSort_v1.

5.6.2 QuickSort v2

En las siguientes gráficas está representado el rendimiento de Mergesort y QuickSort forzándolos con las peores permutaciones para el QuickSort_v2. Aunque sea la misma permutación, los test están hechos con 50 permutaciones por tamaño para suavizar la gráfica de tiempo.



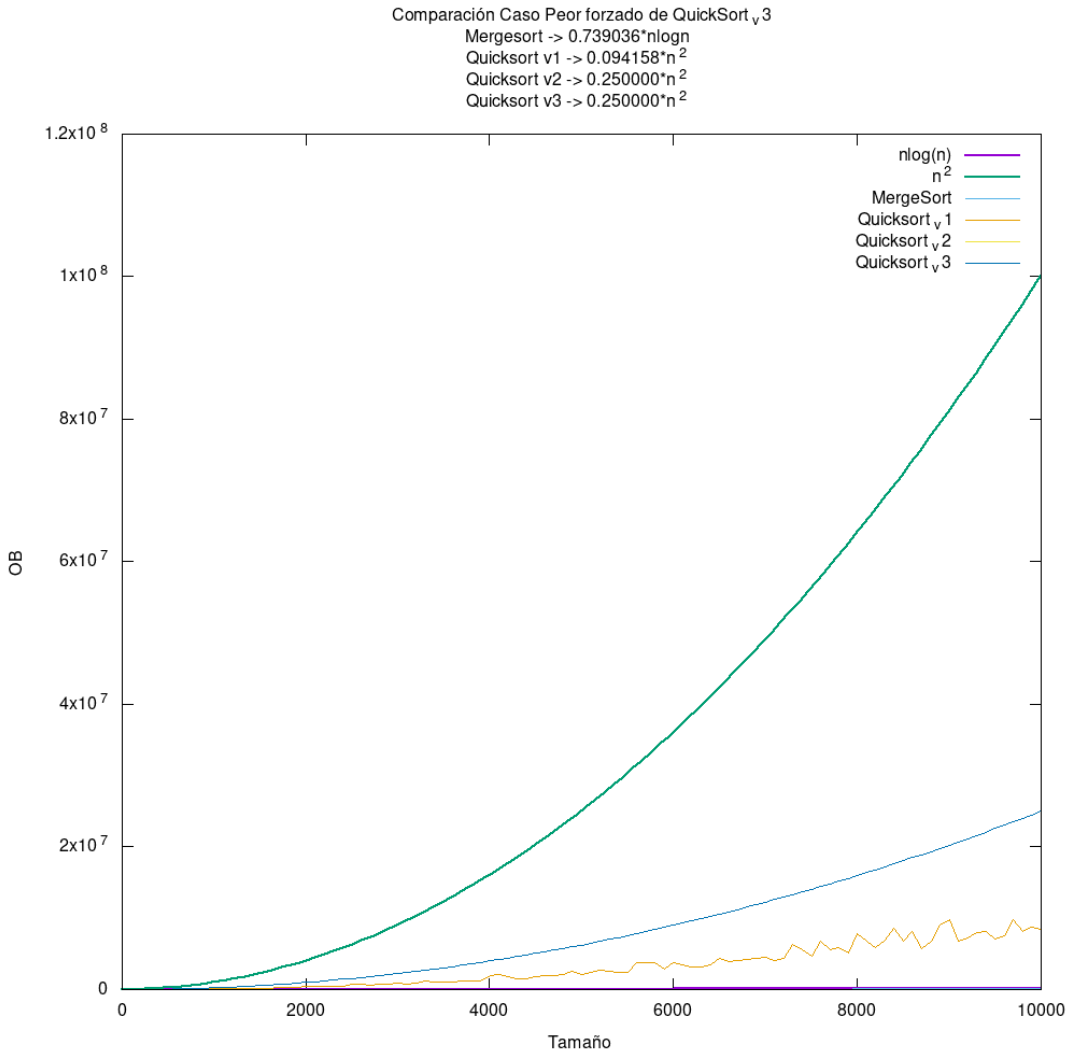
En cuanto a operaciones básicas, el Mergesort siempre va a tener un rendimiento loglineal, lo interesante de este caso es que, a diferencia de con el caso peor del QuickSort_v1, el rendimiento de los QuickSort es cuadrático y además parece seguir un patrón específico en el caso del QuickSort v1 y v3, parece que hay ciertos tamaños que le cuesta significativamente menos ordenarlos y se repiten periódicamente. En cualquier caso, el rendimiento del QuickSort_v1 y v3 sigue siendo entre tres y cuatro veces mejor que el v2.



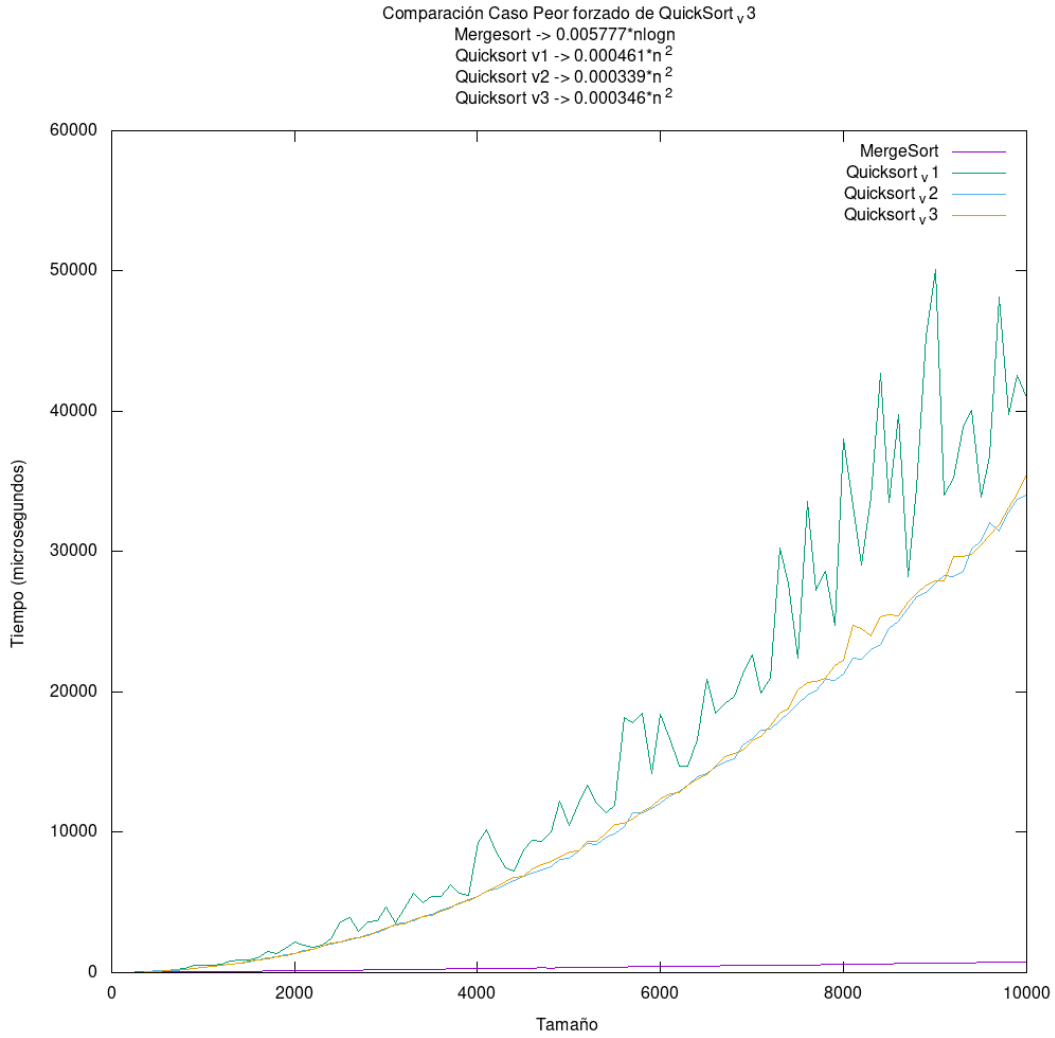
En cuanto a tiempo, el Mergesort desempeña mucho mejor que el Quicksort al ser loglineal. En cambio, los QuickSort tienen un rendimiento cuadrático y como era de esperar, los QuickSort_v1 y v3 parecen seguir un patrón semejante al de las OBs, lo interesante es que el QuickSort_v3 tarda más en ejecutarse que el v2, aunque las OBs sean menores; suponemos que se debe a la mayor complejidad de la función `median_stat` (correspondiente al v3) frente a `median_avg` (correspondiente a v2).

5.6.3 QuickSort v3

En las siguientes gráficas está representado el rendimiento de Mergesort y QuickSort forzándolos con las peores permutaciones para el QuickSort_v3. Aunque sea la misma permutación, los test están hechos con 50 permutaciones por tamaño para suavizar la gráfica de tiempo.



En cuanto a operaciones básicas, el Mergesort siempre va a tener un rendimiento loglineal, lo interesante de este caso es lo siguiente: el QuickSort_v2 y v3 tienen exactamente el mismo rendimiento, aunque sabemos que el peor absoluto del QuickSort_v2 es el doble que el de v3. El rendimiento de QuickSort_v1 es cuadrático, aunque con un comportamiento errático, seguramente debido al extraño patrón de las permutaciones.



En cuanto a tiempo, el Mergesort desempeña mucho mejor que el Quicksort al ser loglineal. Los QuickSort tienen un rendimiento cuadrático, pero lo llamativo aquí es que mientras que QuickSort_v2 y v3 tardan prácticamente lo mismo, el QuickSort_v1 tarda mucho más de lo que debería, ya que en OBs tiene menos de la mitad. No se nos ocurre una explicación razonable para essta diferencia

6. Respuesta a las preguntas teóricas

6.1 Rendimiento empírico vs caso teórico

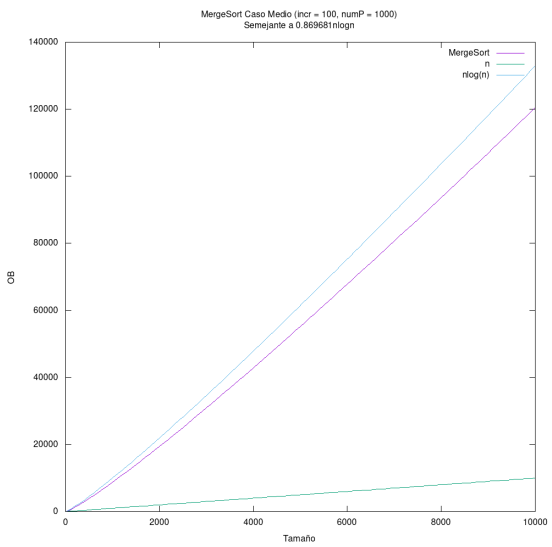
Sabemos por teoría que los casos medios son los siguientes:

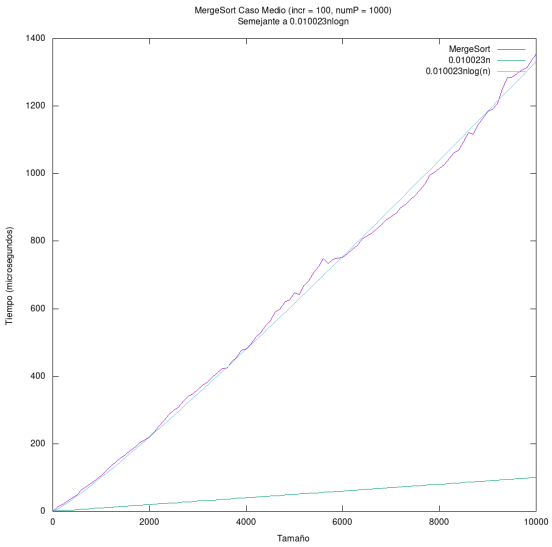
$$A_{MS}(N) = \Theta(N \log N)$$

$$A_{QS}(N) = \Theta(N \log N)$$

Veamos si los resultados empíricos se ajustan a ello.

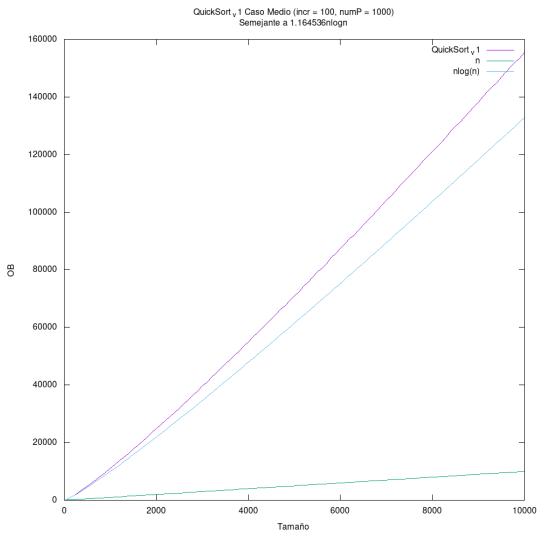
6.1. MergeSort

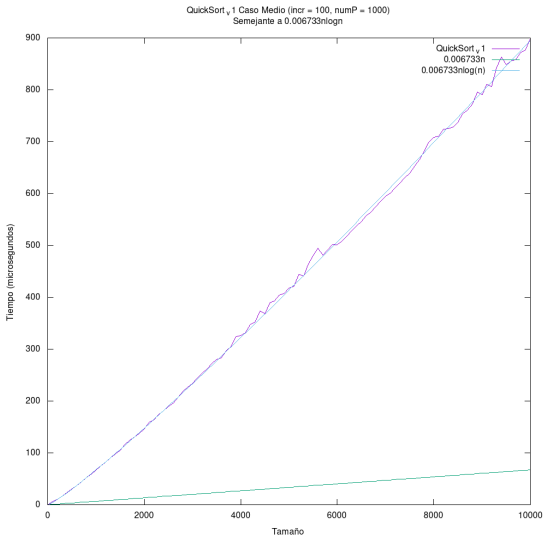




Sí, el rendimiento tanto abstracto como puro es del orden esperado ($\Theta(N \log N)$). El rendimiento puro (en tiempo) es mucho mayor que los de los otros algoritmos en proporción al rendimiento abstracto. Esto es debido a la gestión de la memoria auxiliar.

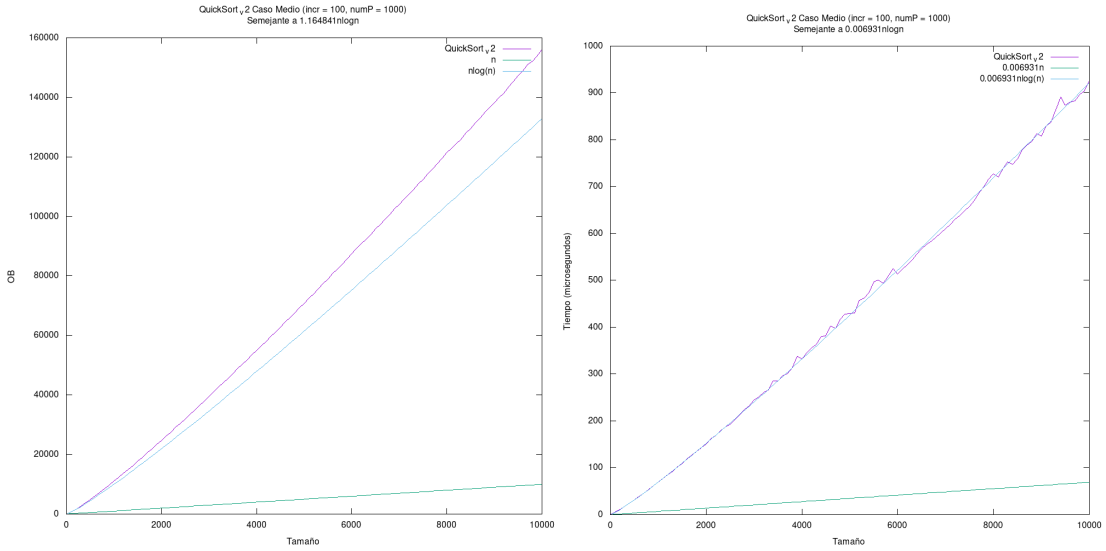
6.1.2 QuickSort_v1





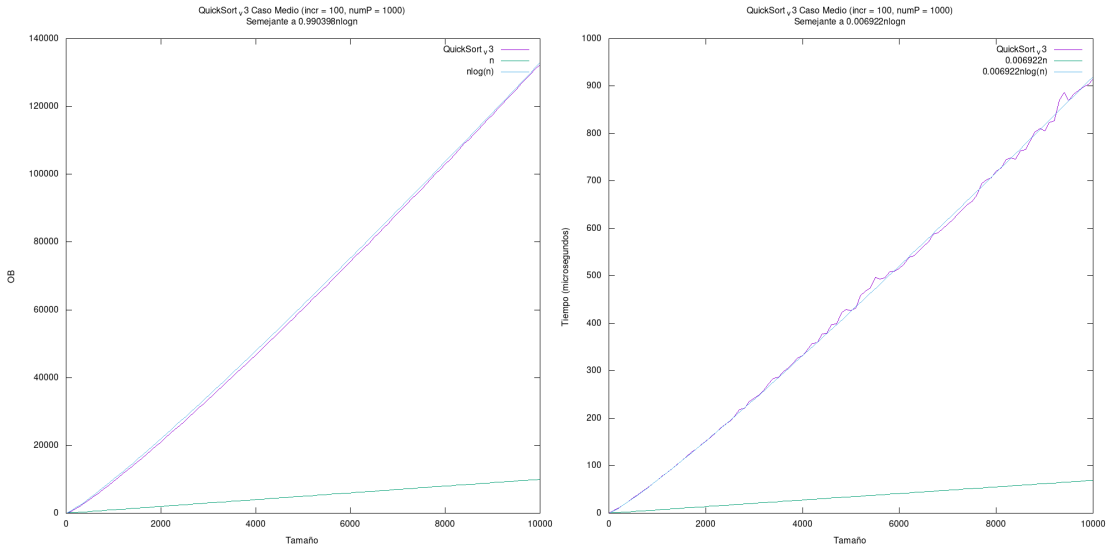
Sí, el rendimiento tanto abstracto como puro es del orden esperado ($\Theta(N \log N)$).

6.1.3 QuickSort_v2



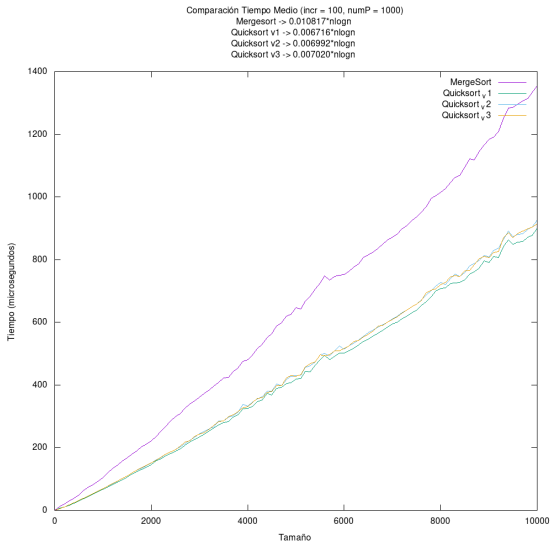
Sí, el rendimiento tanto abstracto como puro es del orden esperado ($\Theta(N \log N)$).

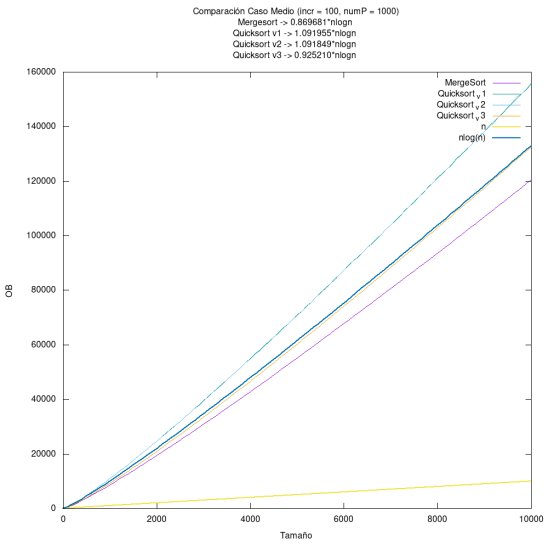
6.1.4 QuickSort_v3



Sí, el rendimiento tanto abstracto como puro es del orden esperado ($\Theta(N \log N)$).

6.2 Diferencias de rendimiento de QuickSort con diferentes pivotes.





En cuanto a operaciones básicas, tomamos como referencia el QuickSort v3 y comprobamos que el QuickSort v1 y v2 son un 18% más costosas. Sin embargo, en cuanto a tiempo son prácticamente idénticas, siendo el QuickSort v1 el más rápido con un 3% de mejora. Esto puede parecer contraintuitivo, ya que el funcionamiento básico de las tres versiones de QuickSort es el mismo, y la elección de pivote de la versión 3 es más inteligente que la 1. Algunos posibles motivos que se nos ocurren son:

- Optimización del compilador: como la función `median` devuelve el primer elemento del array y es extremadamente sencilla, puede que el compilador directamente elimine la función e incluya su funcionalidad en el código del QuickSort_v1, mientras que en el resto de versiones no lo haga y el programa necesite hacer el salto a la función, procesar el pivote y retornar.

- Simplicidad aritmética: otra opción plausible es que simplemente el QuickSort_v1 no hace operaciones en su función `median` y el resto sí, el QuickSort_v2 tiene que calcular el punto medio y el v3 además necesita hacer entre 2 y 3 comparaciones.

En conclusión, en cuanto a operaciones básicas el QuickSort v3 tiene una mejora notable que no se ve traducida en tiempo, de manera que los tres algoritmos tardan esencialmente lo mismo.

6.3 Casos mejor y peor

Tenemos los siguientes rendimientos para los casos mejor y peor, estudiados en clase de teoría:

$$B_{MS}(N) = \Theta(N \log N) \qquad W_{MS}(N) = \Theta(N \log N)$$

$$B_{QS}(N) = \Theta(N \log N) \qquad W_{QS}(N) = \frac{N^2}{2} - \frac{N}{2} = \Theta(N^2)$$

El QuickSort considerado es nuestro QuickSort_v1, pero estos casos nos sirven también para el QuickSort_v2. Sin embargo, QuickSort_v3, tiene un caso peor menor: $W_{QS_3}(N) \approx \frac{N^2}{4}$, aunque el caso mejor sí que es del mismo orden. No podría ser de otra manera, pues el rendimiento loglineal es una cota inferior para los algoritmos de ordenación por CDC (visto también en teoría).

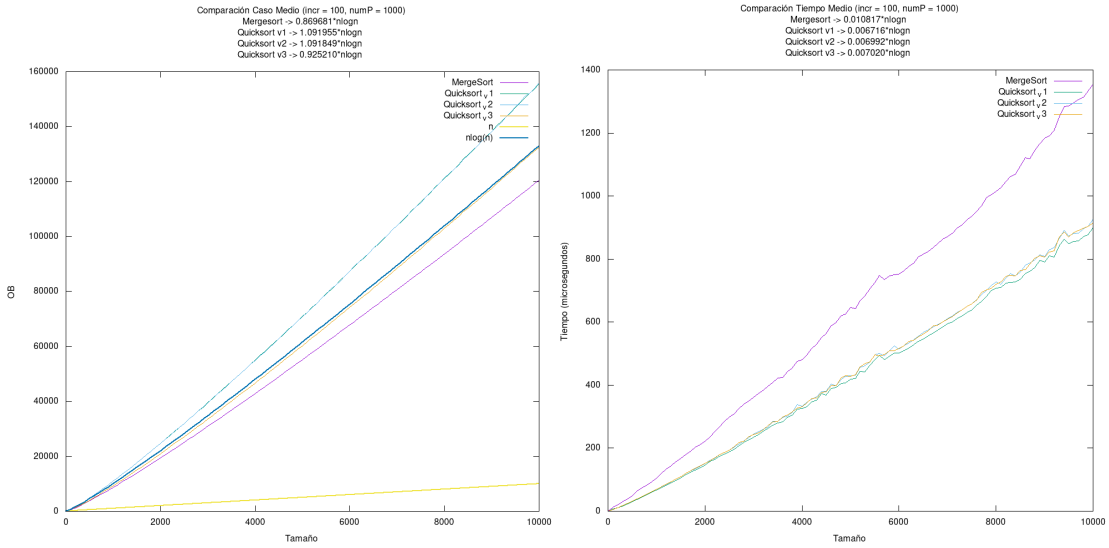
El caso peor ya lo hemos calculado más o menos estrictamente usando las funciones indicadas en los apartados 3.4 y 3.5, y comprobamos que coincide con el teórico observando las gráficas del apartado 5.4. Para terminar de redondearlo, habría que generalizar las funciones `generate_mergesort_worst_perm` y `generate_QuickSort_worst_perm_v3` de forma que puedan generar las permutaciones para cualquier tamaño (no solo potencias de 2 en el primero y pares en el segundo).

Para el caso mejor habría que tomar una estrategia similar, diseñar funciones que generen la mejor permutación para cada tamaño, buscando el objetivo contrario:

- En MergeSort, que en cada llamada a la función de combinar, toda una tabla sea menor que la otra.
- En QuickSort, que en cada partición, el pivote resulte ser la mediana de todos los elementos de la tabla (de esta forma se hará una partición equilibrada).

Por último, si quisiéramos calcular el caso medio de manera estricta, tendríamos que hacer el promedio del costa de todas las permutaciones posibles para cada tamaño N , es decir, $N!$, lo cual es totalmente inasumible computacionalmente.

6.4 Algoritmo más eficiente



Como ya hemos comentado repetidamente, el rendimiento de MergeSort en las pruebas realizadas es mejor en cuanto a OBs, pero en términos temporales QuickSort le aventaja notablemente. En concreto, QuickSort_v1 es el algoritmo más rápido en nuestras pruebas, lo que puede parecer extraño al tener una elección de pivote menos inteligente que QuickSort_v3, pero en el apartado 6.2 explicamos algunos posibles motivos para esto.

En cualquier caso, todos los algoritmos tienen un rendimiento promedio comparable (del mismo orden), como esperábamos del análisis teórico, siendo las tres versiones de QuickSort bastante similares en tiempo de ejecución.

Los casos mejores y peores empíricos también se ajustan a lo esperado teóricamente (lo hemos visto en los apartados 5.4 y 5.5):

$$W_{MS} = \Theta(N \log N) \quad B_{MS} = \Theta(N \log N)$$

$$W_{QS} = \Theta(N^2)\, B_{QS} = \Theta(N \log N)$$

Evidentemente, la gestión de memoria es infinitamente mejor en QuickSort, que es un algoritmo *in place*, así que la memoria utilizada para almacenar la permutación inicialmente es toda la que necesita. El MergeSort, en cambio, reserva memoria en cada combinación para almacenar los elementos de las dos tablas que está combinando.

7. Conclusión

Después de haber comparado los rendimientos de MergeSort y QuickSort en multitud de escenarios, pero siempre para la ordenación de arrays de enteros, consideramos que para este tipo de estructuras de datos casi siempre es preferible utilizar QuickSort, pues es más rápido y no requiere memoria adicional, a menos que sepamos que lo que queremos ordenar suele parecerse mucho al caso peor de QuickSort.

Sin embargo, creemos que puede haber escenarios donde este rendimiento se vea lastrado por otro tipo de problemas. Por ejemplo, en una estructura de lista enlazada, las operaciones de swap pueden ser bastante más costosas, y QuickSort realiza muchos swaps, mientras que MergeSort simplemente va recorriendo secuencialmente la lista al hacer las CDCs, y haciendo una copia de las referencias (en C podría ser un array de punteros). En estos casos, MergeSort podría ahorrar tiempo a cambio del mayor consumo de memoria.

En resumen, ambos algoritmos tienen sus mejores casos de uso, y en el de esta práctica, véase arrays de enteros en C, la mejor opción es QuickSort, aunque esto no debe considerarse motivo para descartar la utilidad de MergeSort en otros contextos.