

PRÁCTICA 1

ALGORÍTMICA

Integrantes:

Raúl Granados López
Hossam El Amraoui Leghzali
Javier Montaña Rubio

1. Algoritmos iterativos

1.1- El funcionamiento de este algoritmo consiste en que, cogida la primera posición del vector (llamémosla *i*), se va comprobando si las posiciones siguientes están repetidas y, en caso afirmativo, se elimina ese número haciendo uso de un bucle. Una vez comprobados todos los elementos siguientes, avanzamos *i* a la posición siguiente y repetimos el procedimiento.

```
void EliminaRepetidos(int* v, int& N){
    int i = 0;
    for (int i = 0; i < N-1; i++){
        int j = i+1;
        while (j < N-1){
            if (v[i] == v[j]){
                for (int k = j; k < N-1; k++){
                    v[k] = v[k+1];
                }
                N--;
            }
            else{
                j++;
            }
        }
    }
}
```

1.2- El funcionamiento de este algoritmo es más simple que el anterior, ya que solo hay que ir comprobando si el elemento siguiente coincide con el elemento cogido (anteriormente llamado *i*), borrarlo en caso afirmativo y seguir realizando la comprobación hasta que el número siguiente sea distinto (obligatoriamente mayor), en cuyo caso avanzaremos *i* y se repite el proceso.

```
void EliminaRepetidos(int* v, int& N){
    int i = 0;
    while (i < N){
        if (v[i] == v[i+1]){
            for (int j = i; j < N-1; j++){
                v[j] = v[j+1];
            }
            N--;
        }
        else{
            i++;
        }
    }
}
```

1.3- Según se puede ver, el algoritmo depende exclusivamente del tamaño del vector (en este caso llamado N), es decir, el tamaño del problema en cada algoritmo sería N.

1.4/1.5-

Primer algoritmo

- Para este algoritmo, el mejor caso posible consistiría en que el vector no tuviese elementos repetidos. Pongamos un ejemplo simple:
 - Imaginemos un vector de 5 elementos sin repetir -> {1,5,8,3,7}
- Al realizar las comprobaciones, cuando entre en el segundo bucle, el algoritmo no tendrá que redimensionar el vector en ningún momento así que no hará falta entrar en el bucle dedicado a ello, mejorando el orden de eficiencia de $O(n^3)$ a $O(n^2)$.

```
82 void EliminaRepetidos(int* v, int& N) {
83
84     int i = 0;  $\rightarrow O(1)$ 
85
86     for (int i = 0; i < N-1; i++) {  $n^o \text{ veces} = n-1$ 
87
88         int j = i+1;  $\rightarrow O(1)$ 
89
90         while (j < N-1) {  $n^o \text{ veces} = n-1$ 
91
92             if (v[i] == v[j]) {  $\rightarrow O(1)$ 
93                 for (int k = j; k < N-1; k++) {
94                     v[k] = v[k+1];
95                 }
96                 N--;
97             }
98             else {
99                 j++;  $\rightarrow O(1)$ 
100             }
101         }
102     }
103 }
104
105
106
107
108 }
```

Handwritten annotations on the code:

- $O(1)$ next to `int i = 0;`
- $n^o \text{ veces} = n-1$ next to the `for` loop.
- $\rightarrow O(1)$ next to `int j = i+1;`
- $n^o \text{ veces} = n-1$ next to the `while` loop.
- $\rightarrow O(1)$ next to the `if` condition.
- A red box highlights the inner `for` loop and the `N--` statement.
- $O(n^2)$ is written on the right side, spanning the `while` loop and the inner `for` loop.
- $O(n)$ is written on the right side, spanning the `while` loop.
- $\rightarrow O(1)$ next to `j++`.
- A red arrow points from the `else` block to the text "nunca entra en el bucle".

- En cambio, el peor caso posible sería que todos los elementos estuviesen repetidos. Veámoslo con un ejemplo:
 - Imaginemos otro vector de 5 elementos repetidos -> {5,5,5,5,5}
- Al realizar las comprobaciones, siempre entrará en el tercer bucle para redimensionar el vector, teniendo que recorrer el bucle en todas las iteraciones, aumentando el tiempo de ejecución.

```

82 void EliminaRepetidos(int* v, int& N) {
83
84     int i = 0;  $\rightarrow O(1)$ 
85
86     for (int i = 0; i < N-1; i++) {  $n^o \text{ veces} = n-1$ 
87
88         int j = i+1;  $\rightarrow O(1)$ 
89
90         while (j < N-1) {  $n^o \text{ veces} = n-1$ 
91
92             if (v[i] == v[j]) {  $\rightarrow O(1)$ 
93                 for (int k = j; k < N-1; k++) {  $n^o \text{ veces} = n-1$ 
94                     v[k] = v[k+1];  $\rightarrow O(1)$ 
95                 }
96             }
97             N--;  $\rightarrow O(1)$ 
98         }
99     }
100     else {
101         j++;  $\rightarrow O(1)$ 
102     }
103 }
104 }
105 }
106 }
107 }
108 }

```

Complexity analysis for the first algorithm:

- The outer loop runs $n-1$ times.
- The inner while loop runs $n-1$ times for each iteration of the outer loop.
- The innermost for loop runs $n-1$ times for each iteration of the while loop.
- The total complexity is $O(n^3)$.

Segundo algoritmo

- Para este algoritmo, el mejor caso posible consistiría en que el vector no tuviese elementos repetidos (teniendo en cuenta que el vector está ordenado). Pongamos un ejemplo simple:
 - Imaginemos un vector de 5 elementos sin repetir $\rightarrow \{1,2,3,4,5\}$
- Al realizar las comprobaciones, el algoritmo no necesitará nunca entrar en el segundo bucle, recorriendo el vector solo una vez, mejorando el orden de eficiencia de $O(n^2)$ a $O(n)$.

```

81 void EliminaRepetidos(int* v, int& N) {
82
83     int i = 0;  $\rightarrow O(1)$ 
84     while (i < N) {  $n^o \text{ veces} = n-1$ 
85
86         if (v[i] == v[i+1]) {
87             for (int j=i; j<N-1; j++) {
88                 v[j] = v[j+1];
89             }
90             N--;
91         }
92         else {
93             i++;  $\rightarrow O(1)$ 
94         }
95     }
96 }
97 }
98 }

```

Complexity analysis for the second algorithm:

- The while loop runs n times.
- The if condition runs $O(1)$ for each iteration.
- The for loop runs $O(n)$ for each iteration of the while loop.
- The total complexity is $O(n^2)$.

Note: The text "nunca entra en el bucle" (never enters the loop) refers to the for loop, indicating that in the best case (no duplicates), the algorithm only runs the while loop once, achieving $O(n)$ complexity.

- En cambio, el peor caso posible sería que todos los elementos estuviesen repetidos. Veámoslo con un ejemplo:
 - Imaginemos otro vector de 5 elementos repetidos -> {5,5,5,5,5}
 Al realizar las comprobaciones, siempre entrará en el segundo bucle para redimensionar el vector, teniendo que recorrer el bucle en todas las iteraciones, aumentando el tiempo de ejecución. Es el mismo caso que en el primer algoritmo.

```

81 void EliminaRepetidos(int* v, int& N) {
82
83     int i = 0;  $\rightarrow O(1)$ 
84     while (i < N) {  $n^{\circ} \text{ veces} = n-1$ 
85
86          $O(n)$ 
87         if (v[i] == v[i+1]) {
88              $O(n)$ 
89             for (int j=i; j<N-1; j++) {  $n^{\circ} \text{ veces} = n-1$ 
90                 v[j] = v[j+1];  $\rightarrow O(1)$ 
91             }
92             N--;  $\rightarrow O(1)$ 
93         }
94         else {
95             i++;  $\rightarrow O(1)$ 
96         }
97     }
98 }

```

$O(n^2)$

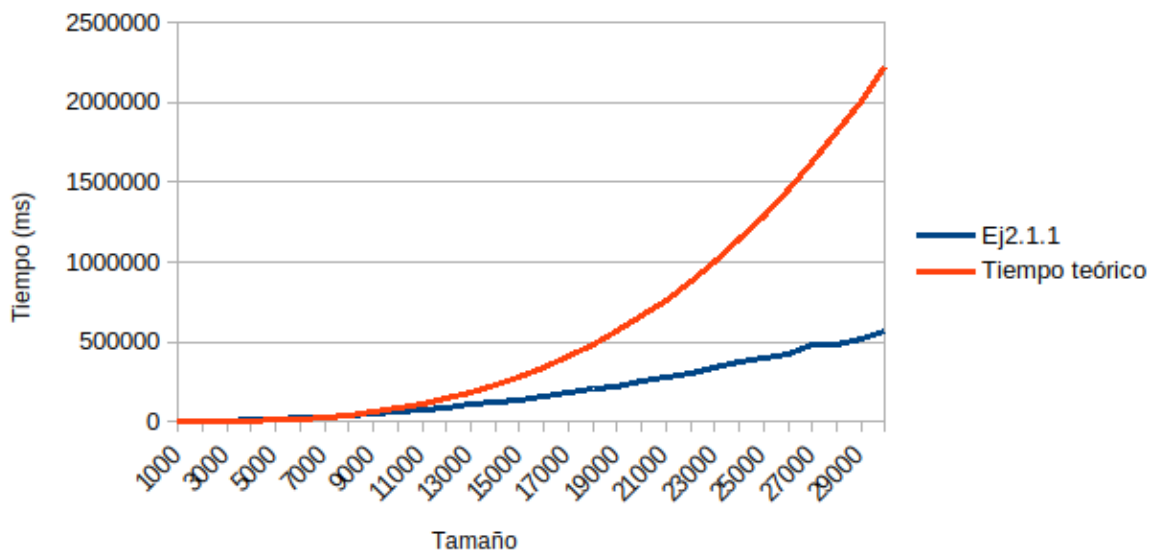
1.6-

Primer algoritmo:

Como se ve, los resultados de la ejecución real son mejores que en la ejecución teórica debido a que la ejecución teórica se calcula para el peor de los casos, por lo que es un hecho que el tiempo de ejecución calculado con una constante oculta siempre será mayor que el tiempo de ejecución real del propio algoritmo.

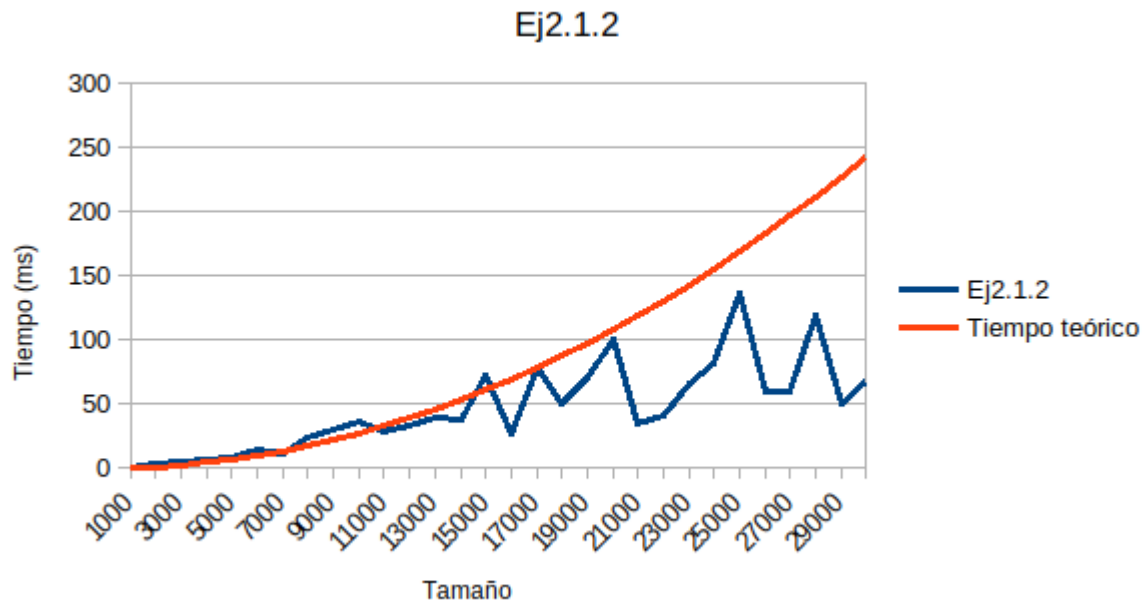
| Tamaño del vector | Ej2.1.1 | $O(n^2)$ | K | Tiempo teórico |
|-------------------|---------|-------------|-------------|------------------|
| 1000 | 622 | 1000000000 | 0,000000622 | 82,55852812353 |
| 2000 | 2330 | 8000000000 | 2,9125E-07 | 660,46822498824 |
| 3000 | 5539 | 27000000000 | 2,05148E-07 | 2229,08025933531 |
| 4000 | 10363 | 64000000000 | 1,61922E-07 | 5283,74579990592 |
| 5000 | 16052 | 1,25E+11 | 1,28416E-07 | 10319,8160154413 |
| 6000 | 22069 | 2,16E+11 | 1,02171E-07 | 17832,6420746825 |
| 7000 | 30436 | 3,43E+11 | 8,87347E-08 | 28317,5751463708 |
| 8000 | 38535 | 5,12E+11 | 7,52637E-08 | 42269,9663992474 |
| 9000 | 51022 | 7,29E+11 | 6,9989E-08 | 60185,1670020534 |
| 10000 | 61251 | 1E+12 | 6,1251E-08 | 82558,52812353 |
| 11000 | 74064 | 1,331E+12 | 5,56454E-08 | 109885,400932418 |
| 12000 | 91233 | 1,728E+12 | 5,27969E-08 | 142661,13659746 |
| 13000 | 105231 | 2,197E+12 | 4,78976E-08 | 181381,086287395 |
| 14000 | 123952 | 2,744E+12 | 4,5172E-08 | 226540,601170966 |
| 15000 | 138857 | 3,375E+12 | 4,11428E-08 | 278635,032416914 |
| 16000 | 162038 | 4,096E+12 | 3,95601E-08 | 338159,731193979 |
| 17000 | 187266 | 4,913E+12 | 3,81164E-08 | 405610,048670903 |
| 18000 | 202679 | 5,832E+12 | 3,47529E-08 | 481481,336016427 |
| 19000 | 221457 | 6,859E+12 | 3,22871E-08 | 566268,944399292 |
| 20000 | 253206 | 8E+12 | 3,16508E-08 | 660468,22498824 |
| 21000 | 275860 | 9,261E+12 | 2,97873E-08 | 764574,528952011 |
| 22000 | 303654 | 1,0648E+13 | 2,85175E-08 | 879083,207459347 |
| 23000 | 337225 | 1,2167E+13 | 2,77164E-08 | 1004489,61167899 |
| 24000 | 374058 | 1,3824E+13 | 2,70586E-08 | 1141289,09277968 |
| 25000 | 393819 | 1,5625E+13 | 2,52044E-08 | 1289977,00193016 |
| 26000 | 425892 | 1,7576E+13 | 2,42315E-08 | 1451048,69029916 |
| 27000 | 488311 | 1,9683E+13 | 2,48088E-08 | 1624999,50905544 |
| 28000 | 481421 | 2,1952E+13 | 2,19306E-08 | 1812324,80936773 |
| 29000 | 522291 | 2,4389E+13 | 2,1415E-08 | 2013519,94240477 |
| 30000 | 564793 | 2,7E+13 | 2,09183E-08 | 2229080,2593354 |
| | | | 8,25585E-08 | |

Ej2.1.1



Segundo algoritmo: Al igual que en el primer algoritmo, los tiempos de ejecución para cada tamaño del vector generalmente son inferiores respecto al tiempo teórico calculado.

| Tamaño del vector | Ej2.1.2 | $O(n^2)$ | K | Tiempo teórico |
|-------------------|---------|-----------|-------------|-------------------|
| 1000 | 1 | 1000000 | 0,000001 | 0,270418444256032 |
| 2000 | 3 | 4000000 | 0,00000075 | 1,08167377702412 |
| 3000 | 5 | 9000000 | 5,55556E-07 | 2,43376599830427 |
| 4000 | 7 | 16000000 | 4,375E-07 | 4,32669510809648 |
| 5000 | 8 | 25000000 | 0,00000032 | 6,76046110640075 |
| 6000 | 15 | 36000000 | 4,16667E-07 | 9,73506399321708 |
| 7000 | 11 | 49000000 | 2,2449E-07 | 13,2505037685455 |
| 8000 | 24 | 64000000 | 0,000000375 | 17,3067804323859 |
| 9000 | 30 | 81000000 | 3,7037E-07 | 21,9038939847384 |
| 10000 | 36 | 100000000 | 0,00000036 | 27,041844425603 |
| 11000 | 29 | 121000000 | 2,39669E-07 | 32,7206317549796 |
| 12000 | 33 | 144000000 | 2,29167E-07 | 38,9402559728683 |
| 13000 | 39 | 169000000 | 2,30769E-07 | 45,7007170792691 |
| 14000 | 38 | 196000000 | 1,93878E-07 | 53,0020150741819 |
| 15000 | 72 | 225000000 | 0,00000032 | 60,8441499576067 |
| 16000 | 27 | 256000000 | 1,05469E-07 | 69,2271217295437 |
| 17000 | 79 | 289000000 | 2,73356E-07 | 78,1509303899927 |
| 18000 | 51 | 324000000 | 1,57407E-07 | 87,6155759389537 |
| 19000 | 70 | 361000000 | 1,93906E-07 | 97,6210583764268 |
| 20000 | 100 | 400000000 | 0,00000025 | 108,167377702412 |
| 21000 | 35 | 441000000 | 7,93651E-08 | 119,254533916909 |
| 22000 | 41 | 484000000 | 8,47107E-08 | 130,882527019919 |
| 23000 | 66 | 529000000 | 1,24764E-07 | 143,05135701144 |
| 24000 | 82 | 576000000 | 1,42361E-07 | 155,761023891473 |
| 25000 | 137 | 625000000 | 2,192E-07 | 169,011527660019 |
| 26000 | 60 | 676000000 | 8,87574E-08 | 182,802868317076 |
| 27000 | 59 | 729000000 | 8,09328E-08 | 197,135045862646 |
| 28000 | 120 | 784000000 | 1,53061E-07 | 212,008060296727 |
| 29000 | 51 | 841000000 | 6,06421E-08 | 227,421911619321 |
| 30000 | 68 | 900000000 | 7,55556E-08 | 243,376599830429 |
| | | | 2,70418E-07 | |



Compilación y ejecución: Para compilar los distintos programas se utilizará el compilador g++ o gcc para obtener sus respectivos ejecutables. En cuanto a su ejecución, será necesario pasarle por la terminal dos argumentos: el tamaño del vector y la semilla de la generación de números aleatorios con los que se rellenará dicho vector.

Para el primer apartado de los algoritmos iterativos:

- Compilación:


```
g++ ej2.1.1.cpp -o ej2.1.1
```
- Ejecución:


```
./ej2.1.1 <N> <Semilla>
```

por ejemplo, tamaño 10, semilla del generador de aleatorios 0:

```
./ej2.1.1 10 0
```

Para el segundo apartado de los algoritmos iterativos:

- Compilación:


```
g++ ej2.1.2.cpp -o ej2.1.1
```
- Ejecución:


```
./ej2.1.2 <N> <Semilla>
```

por ejemplo, tamaño 10, semilla del generador de aleatorios 0:

```
./ej2.1.2 10 0
```

2. Algoritmos recursivos

2.1-

- Para Hanoi:

```

1  /**
2  Se trata del problema clásico de las torres de Hanoi.
3  Se tienen 3 barras, y hay que mover M anillos de la primera barra
4  a la segunda. Solo se puede mover un anillo en cada movimiento,
5  y ningún anillo de tamaño mayor puede ponerse sobre otro de tamaño
6  menor.
7  Los valores de "i" y "j" sólo pueden tomar los valores {1, 2, 3}
8
9  Si M=3, la llamada sería hanoi(3, 1, 2)
10
11  */
12 void hanoi (int M, int i, int j)
13 {
14     if (M > 0) → O(1)
15     {
16         hanoi(M-1, i, 6-i-j); → O(n)
17         cout << i << " -> " << j << endl; → O(1)
18         hanoi (M-1, 6-i-j, j); → O(n)
19     }
20 }

```

$$\begin{aligned}
 T(n) &= 2 \cdot T(n-1) + 1 \\
 T(n) - 2 \cdot T(n-1) &= 1 \\
 \text{PH} \left[\begin{aligned} X^n - 2 \cdot X^{n-1} &= 0 \\ X \cdot X^{n-1} - 2 \cdot X^{n-1} &= 0 \end{aligned} \right. \\
 \Rightarrow 1 = b_1^m \cdot g_1(n) &\left\{ \begin{aligned} b &= 1 \\ g_1(n) &= 1 \end{aligned} \right.
 \end{aligned}$$

$$\begin{aligned}
 X^{n-1} \cdot (X-2) &= 1 \\
 P_H(X) &= (X-2) \\
 p(x) &= P_H(x) \cdot (x-b_1)^{q_1} \\
 p(x) &= (x-2) \cdot (x-1)^1 \\
 r &= 2 \\
 R_1 &= 2 \quad m=1 \\
 R_2 &= 1 \quad m=1
 \end{aligned}$$

$$\begin{aligned}
 T(n) &= c_{10} \cdot R_1^n \cdot n^m + c_{20} \cdot R_2^n \cdot n^m \\
 &= c_{10} \cdot 2^n + c_{20} \cdot 1^n \Rightarrow \\
 &\Rightarrow O(2^n)
 \end{aligned}$$

Hanoi

- Para HeapSort:

```

void insertarEnPos(double *apo, int pos){ → O(log₂(n))
    int idx = pos-1; → O(1)
    int padre; → O(1)
    if (idx > 0) {
        if (idx%2==0) {
            padre=(idx-2)/2; → padre = (n-2)/2
        }else{
            padre=(idx-1)/2;
        }
    }

    if (apo[padre] > apo[idx]) {
        double tmp=apo[idx];
        apo[idx]=apo[padre];
        apo[padre]=tmp;
        insertarEnPos(apo, padre+1); → T(n/2)
    }
}

```

$$T(n/2) + 1$$

Insertar en pos y reestructurar raíz

$$T(u) = T(u/2) + 1$$

$$T(u) - T(u/2) = 1$$

Cambio var: $u = 2^m \rightarrow m = \log_2(u)$

$$T(2^m) - T(2^{m/2}) = 1$$

PH

$$x^m - x^{m-1} = 0$$

$$x^{m-1} \cdot (x-1) = 0$$

$$P_H(x) = (x-1)$$

PNH

$$1 = \begin{cases} b_1 = 1 \\ q_1(m) = 1 \end{cases}$$

$$P(x) = (x-1) \cdot (x-1) = (x-1)^2$$

$$r = 1$$

$$R_1 = 1 \quad m = 2$$

$$T(2^m) = c_p \cdot 1^{\log_2(u)} + c_H \cdot 1^{\log_2(u)} \Rightarrow \Rightarrow O(\log_2(u))$$

$$O(u \cdot \log_2(u))$$

$$O(u \cdot \log_2(u))$$

```
void HeapSort(int *v, int n){  $\rightarrow O(u \cdot \log_2(u))$ 
    double *apo=new double [n];
    int tamapo=0;

    for (int i=0; i<n; i++){  $u^o$  veces  $\sim N$ 
        apo[tamapo]=v[i];
        tamapo++;
        insertarEnPos(apo,tamapo);  $\rightarrow O(\log_2(u))$ 
    }

    for (int i=0; i<n; i++){  $u^o$  veces  $\sim u$ 
        v[i]=apo[0];  $\rightarrow O(1)$ 
        tamapo--;  $\rightarrow O(1)$ 
        apo[0]=apo[tamapo];  $\rightarrow O(1)$ 
        reestructurarRaiz(apo, 0, tamapo);  $\rightarrow O(\log_2(u))$ 
    }

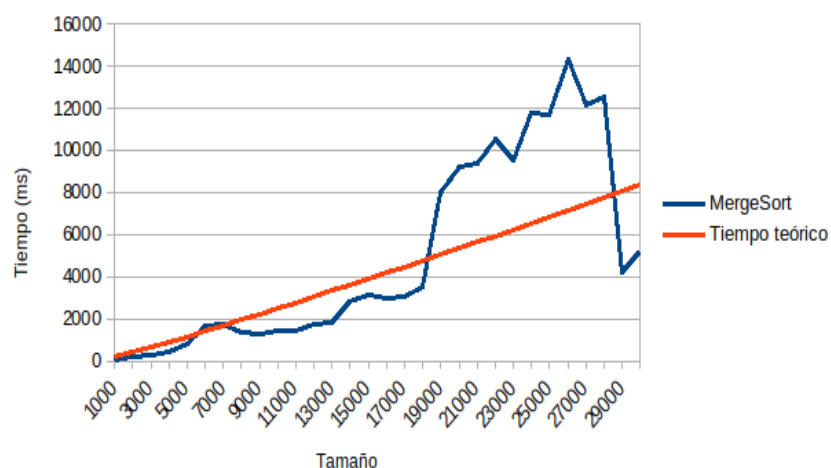
    delete [] apo;
}
```

2.2-

MergeSort

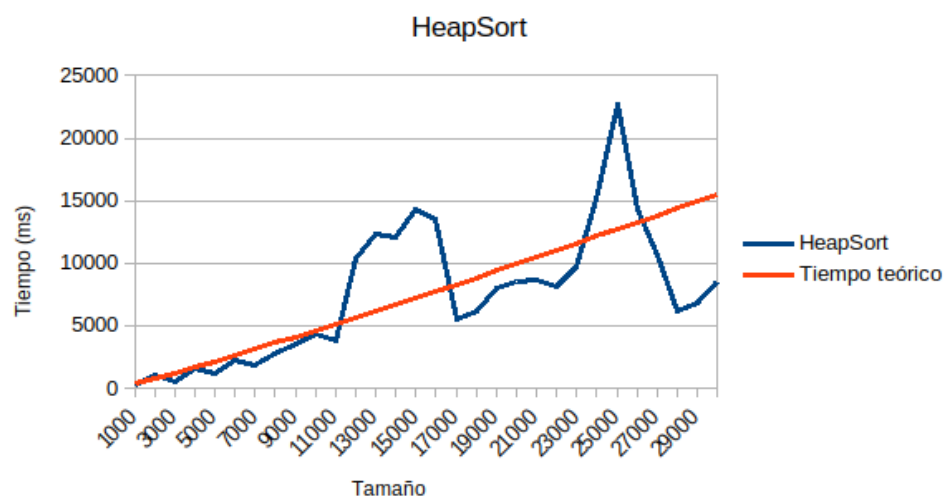
| Tamaño del vector | MergeSort | $O(n \cdot \log_2(n))$ | K | Tiempo teórico |
|-------------------|-----------|------------------------|-------------|------------------|
| 1000 | 77 | 9965,784285 | 0,007726437 | 186,292585736891 |
| 2000 | 160 | 21931,56857 | 0,00729542 | 409,971608991521 |
| 3000 | 250 | 34652,24036 | 0,007214541 | 647,761909462429 |
| 4000 | 461 | 47863,13714 | 0,009631629 | 894,71609301852 |
| 5000 | 824 | 61438,5619 | 0,013411772 | 1148,48447778493 |
| 6000 | 1659 | 75304,48071 | 0,022030562 | 1407,68313147807 |
| 7000 | 1741 | 89411,97445 | 0,019471665 | 1671,39759800092 |
| 8000 | 1318 | 103726,2743 | 0,01270652 | 1938,977936108 |
| 9000 | 1250 | 118221,3836 | 0,010573383 | 2209,93818514256 |
| 10000 | 1403 | 132877,1238 | 0,010558627 | 2483,90114315855 |
| 11000 | 1395 | 147677,3749 | 0,009446268 | 2760,56547541971 |
| 12000 | 1733 | 162608,9614 | 0,010657469 | 3039,68488806258 |
| 13000 | 1829 | 177660,912 | 0,010294893 | 3321,05429359745 |
| 14000 | 2832 | 192823,9489 | 0,014686972 | 3604,50025862601 |
| 15000 | 3098 | 208090,1232 | 0,01488778 | 3889,87419461356 |
| 16000 | 2973 | 223452,5486 | 0,013304838 | 4177,0473723579 |
| 17000 | 3073 | 238905,2011 | 0,012862843 | 4465,90718755127 |
| 18000 | 3509 | 254442,7671 | 0,013790921 | 4756,35430794476 |
| 19000 | 8013 | 270060,5242 | 0,029671127 | 5048,30045633273 |
| 20000 | 9205 | 285754,2476 | 0,032212994 | 5341,66666149449 |
| 21000 | 9360 | 301520,1359 | 0,031042703 | 5636,38185976505 |
| 22000 | 10508 | 317354,7499 | 0,033111211 | 5932,38176353454 |
| 23000 | 9554 | 333254,9635 | 0,02866874 | 6229,60793586757 |
| 24000 | 11766 | 349217,9228 | 0,033692429 | 6528,00702633803 |
| 25000 | 11698 | 365241,0119 | 0,032028167 | 6827,530134427 |
| 26000 | 14328 | 381321,8241 | 0,037574561 | 7128,13227492551 |
| 27000 | 12129 | 397458,1382 | 0,030516421 | 7429,77192569347 |
| 28000 | 12549 | 413647,8978 | 0,030337396 | 7732,41064250036 |
| 29000 | 4170 | 429889,1931 | 0,009700174 | 8036,01272895199 |
| 30000 | 5215 | 446180,2464 | 0,011688101 | 8340,54495199321 |
| | | | 0,018693219 | |

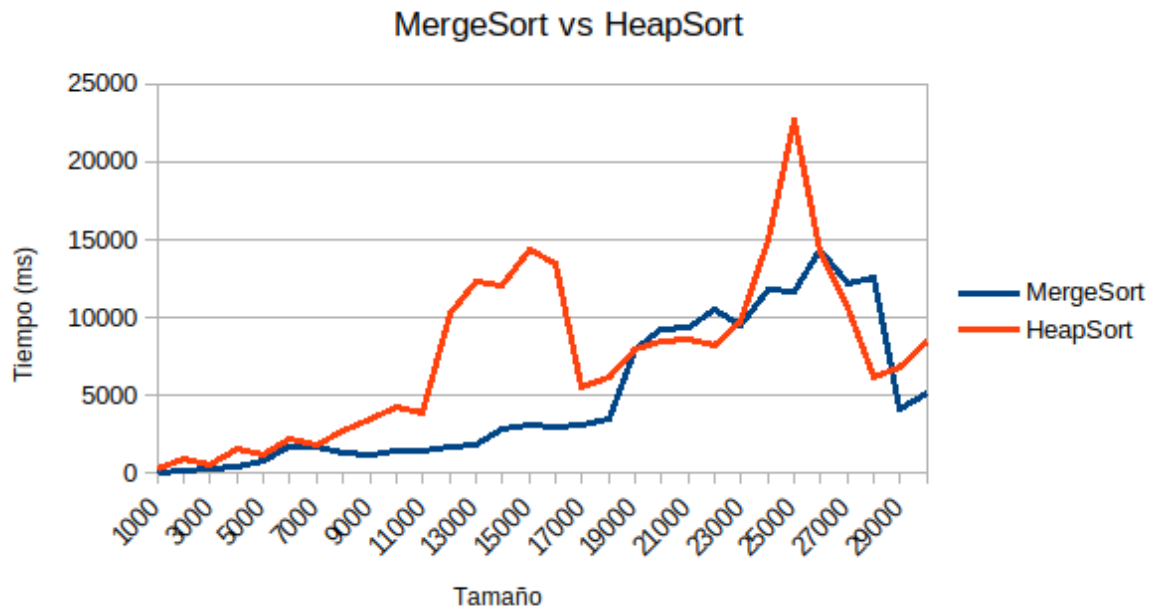
MergeSort



HeapSort

| Tamaño del vector | HeapSort | $O(n \cdot \log_2(n))$ | K | Tiempo teórico |
|-------------------|----------|------------------------|-------------|------------------|
| 1000 | 319 | 9965,784285 | 0,032009523 | 346,817580394768 |
| 2000 | 1000 | 21931,56857 | 0,045596374 | 763,236823937824 |
| 3000 | 505 | 34652,24036 | 0,014573372 | 1205,92678030109 |
| 4000 | 1623 | 47863,13714 | 0,033909186 | 1665,67697417222 |
| 5000 | 1226 | 61438,5619 | 0,019954894 | 2138,11304476107 |
| 6000 | 2200 | 75304,48071 | 0,029214729 | 2620,65855004705 |
| 7000 | 1809 | 89411,97445 | 0,020232189 | 3111,61106344296 |
| 8000 | 2721 | 103726,2743 | 0,026232505 | 3609,76060093758 |
| 9000 | 3545 | 118221,3836 | 0,029986115 | 4114,20245825365 |
| 10000 | 4282 | 132877,1238 | 0,032225261 | 4624,23440526358 |
| 11000 | 3833 | 147677,3749 | 0,025955228 | 5139,29545246954 |
| 12000 | 10310 | 162608,9614 | 0,06340364 | 5658,92707898382 |
| 13000 | 12281 | 177660,912 | 0,069126066 | 6182,74747708887 |
| 14000 | 12043 | 192823,9489 | 0,062455935 | 6710,43376892392 |
| 15000 | 14324 | 208090,1232 | 0,068835559 | 7241,70932986717 |
| 16000 | 13492 | 223452,5486 | 0,06037971 | 7776,33450706146 |
| 17000 | 5535 | 238905,2011 | 0,023168185 | 8314,09966707774 |
| 18000 | 6195 | 254442,7671 | 0,024347322 | 8854,81988484188 |
| 19000 | 7999 | 270060,5242 | 0,029619286 | 9398,33081625647 |
| 20000 | 8542 | 285754,2476 | 0,029892819 | 9944,48544201002 |
| 21000 | 8661 | 301520,1359 | 0,02872445 | 10493,1514641464 |
| 22000 | 8186 | 317354,7499 | 0,025794478 | 11044,2091995702 |
| 23000 | 9763 | 333254,9635 | 0,029295888 | 11597,5498572823 |
| 24000 | 15055 | 349217,9228 | 0,043110617 | 12153,0741157471 |
| 25000 | 22652 | 365241,0119 | 0,062019322 | 12710,6909377415 |
| 26000 | 14282 | 381321,8241 | 0,037453928 | 13270,3165751055 |
| 27000 | 10613 | 397458,1382 | 0,026702183 | 13831,8737268121 |
| 28000 | 6182 | 413647,8978 | 0,014945078 | 14395,2908219239 |
| 29000 | 6817 | 429889,1931 | 0,015857575 | 14960,5014051012 |
| 30000 | 8479 | 446180,2464 | 0,019003531 | 15527,4436069586 |
| | | | 0,034800832 | |





Como se puede observar, aún habiendo en ciertas ocasiones claras diferencias entre el tiempo transcurrido, se podría decir que en bastantes ocasiones se comportan de manera similar. Las diferencias pueden venir dadas por la diferencia entre las constantes ocultas, siendo, de media, mayores las del HeapSort que las del MergeSort, aún teniendo ambos algoritmos el mismo orden de eficiencia.