Practica 1

Eficiencia de algoritmo

Francisco Javier Merchán

Eficiencia Teórica

Eficiencia Algoritmos no Recurrentes

a)

```
int pivotar(double *v, const int ini, const int fin) {

    double pivote= v[ini], aux;
    int i= ini+1, j= fin;


    while (i<=j) {
        while (v[i]<pivote && i<=j) i++;
        while (v[j]>=pivote && j>=i) j--;

        if (i<j) {
            aux= v[i]; v[i]= v[j]; v[j]= aux;
        }
    }


    if (j>ini) {
        v[ini]= v[j];
        v[j]= pivote;
    }
    return j;
}
```

Este algoritmo resuelve el problema de ordenación de un vector, ordena con un criterio de pivote, dejando a la izquierda los menores y la derecha los mayores. El tamaño del caso es n el cual representa:

n=fin-ini

La función está formada por 21 líneas de código entre las que se encuentran varias sentencias simples al principio del programa (declaraciones e inicializaciones), específicamente 3, las cuales tienen eficiencia O(1). A continuación, encontramos un bucle while que va desde i, que es ini+1, hasta j, que una posición final pasado como parámetro.

Para saber la eficiencia del while analizaremos antes el cuerpo. El cuerpo está formado por dos bucles while y una condición if. La condición if sin dudad es O(1) ya que le max(O(1),O(1)), son la eficiencia del cuerpo y de la condición respectivamente, es O(1). Lo importante de la eficiencia del bucle while exterior son los dos bucles while que hay dentro, los cuales el peor caso de cada uno discrimina al otro ya que si el pivote es el menor de todos recorre todo hacia la izq y si es mayor al revés, pero como no hay ningún número que sea mayor y menor a la vez que otro pues solo se puede ejecutar uno de ellos.

Por lo tanto, por lo que sabes de teoría es que la eficiencia del bucle while es (función del bucle while) que es igual a O(n), ya que el bucle que se haga como peor caso dará n vueltas si suponemos que el pivote sea el mayor de todos en el primer caso o menor, si decimos que el peor caso es el del segundo while.

Después del bucle while encontramos una condición if, la cual tiene como eficiencia el máximo del cuerpo y de la condición como hemos dicho antes, es decir, max(O(1),O(1)), que es O(1).

La suma de todas las ‘O’ podemos decir que la eficiencia de este algoritmo en el peor de los casos es O(n).

En el mejor de lo casos, pasa lo mismo, sigue necesitando recorrer el vector entero por lo tanto su eficiencia será O(n), y podemos decir que es una eficiencia exacta ya que en el peor y en el mejor caso es la misma.

b)

```
int Busqueda (int *v, int n, int elem) {

  int inicio, fin, centro;

  inicio= 0;
  fin= n-1;
  centro= (inicio+fin)/2;
  while ((inicio<=fin) && (v[centro] != elem)) {

        if (elem<v[centro])
              fin= centro-1;
        else
              inicio= centro+1;
         centro= (inicio+fin)/2;
}

  if (inicio>fin)
        return -1;

  return centro;
}
```

Este algoritmo resuelve el problema de buscar un elemente en un vector. Para ello nos da n que el tamaño del vector en el que tenemos que buscar el elemento (elem) deseado. Con la condición de que ese vector este ordenado de menor a mayor.

El tamaño del vector también será el tamaño del problema al que nos enfrentemos, en este caso será n.

Comenzaremos analizando el algoritmo por las primeras líneas, las cuales son todas operaciones elementales, declaraciones y inicializaciones, todas ellas de eficiencia O(1). Seguidamente vemos un bucle while, el cual como sabemos por teoría su eficiencia viene definida por O(g(n) + h(n)*(g(n)+f(n))).

Dentro del bucle while encontramos un if junto con un else, ambos O(1), ya que su interior y la condición del if son operaciones elementales, en este caso comparaciones, y la eficiencia máxima de esas 3(max{O(1), O(1), O(1)}), es O(1). Después nos encontramos una asignación que como es una operación elemental es O(1). Por lo tanto el cuerpo del bucle while tiene una eficiencia de O(1).

El bucle while se ejecuta n/2 veces, ya que como vemos las condiciones de salida es que fin sea igual o mayor, es decir que el fin sea menor que inicio, o que el centro de la parte del vector en la que te encuentres sea igual al elemento que estás buscando. Como podemos comprobar en el cuerpo del bucle en la sentencia condicional if, si el elemento es menor que el central de esa parte del vector busca en la parte izquierda, si no busca en la derecha. Como en cada iteración te quitas la mitad acabas recorriendo solo la mitad de los elementos. Por lo tanto, podemos decir que tiene una eficiencia de O(n/2).

Después del bucle while no encontramos con una sentencia condicional if la cual tiene como cuerpo una operación elemental, cuya eficiencia es O(1). La eficiencia de este if ser el max entre el cuerpo y la condición, es decir, max{{O(1), O(1)}, que es O(1).

Podemos que decir que la eficiencia de este algoritmo será definida por el bucle while ya que es la max entre sus eficacias, todas ellas O(1). El algoritmo tendrá en el peor de los casos, que en una búsqueda es que el no este elemento donde se esté buscando, de O(n/2).

En mejor de los casos será O(1), ya que el elemento estará situado en el centro y el while solo hará la comparación.

c)

```
void EliminaRepetidos(double original[], int & nOriginal) {

 int i, j, k;

 // Pasamos por cada componente de original
 for (i= 0; i<nOriginal; i++) {

        // Buscamos valor repetido de original[i]
        // desde original[i+1] hasta el final
           j= i+1;
           do {

                 if (original[j] == original[i]) {

                   // Desplazamos todas las componentes desde j+1
                   // hasta el final, una componente a la izquierda
                   for (k= j+1; k<nOriginal; k++)
                          original[k-1]= original[k];

                   // Como hemos eliminado una componente, reducimos
                   // el numero de componentes utiles
                   nOriginal--;
                 } else // Si el valor no esta repetido, pasamos al siguiente j
                   j++;
           } while (j<nOriginal);

   } // FIN del primer for
  }
```

Este algoritmo da una solución para el problema de elementos eliminar los repetidos de un vector.

El tamaño del problema es de n, que es el tamaño del vector pasado como parámetro (nOriginal).

La eficiencia de este algoritmo vendrá definida por la eficiencia del bucle for, antes de este hay una declaraciones de variables que son O(1). Para el bucle for y como sabemos por teoría es **O(i(n)+g(n) + h(n)*(g(n)+f(n)+a(n)))** .

Dentro del bucle for, encontramos una inicialización la cual es una operación elemental, es decir su eficiencia es de O(1). Seguidamente vemos un do-while el cual como teoría vimos en teoría su eficiencia está definida por **O(f(n) + g(n) + h(n)*(f(n)+g(n)))**.

El cuerpo del do-while está compuesto por una sentencia condicional if-else. El if tiene como eficiencia O(n) en el peor de los casos, y el else como solo tiene una operación elemental es O(1). En teoría vimos que la eficiencia de una sentencia condicional if-else es max{O(1),O(n),O(1)}, es decir O(n). Es O(n) porque el bucle for que hay dentro del if es O(n) ya que recorre todos los elementos del vector.

Lo interesante es que que si en el bucle do-while damos como peor caso el if, al ir decrementando el tamaño del vector nos encontramos que el primer bucle for solo da una vuelta, es decir que sería su mejor caso. Dando como resultado un algoritmo O(n^2). Pero si damos como el mejor caso en el if, es decir que se cumpla el else, el primer for estará en su peor caso dando un algoritmo O(n^2). Este caso es como los dos bucles while de la función *"pivotar"* de esta práctica.

Por lo que podemos decir que la suma de estos, cualquiera de ambos casos seria O(n^2). Como en el mejor de los casos y el peor de los casos se entre mezclan, las sumas en ambos casos serán O(n^2), por lo que el algoritmo es de orden exacto.

Eficiencia Algoritmos Recurrentes

a)

```
int BuscarBinario(double *v, const int ini, const int fin,
                  const double x) {
    int centro;
    if (ini>fin) return -1;

    centro= (ini+fin)/2;
    if (v[centro] == x) return centro;
    if (v[centro]>x) return BuscarBinario(v, ini, centro-1, x);
    return BuscarBinario(v, centro+1, fin, x);
}
```

Este algoritmo da una solución recursiva al problema de la búsqueda, una versión recursiva del de búsqueda visto anteriormente. El vector tiene que estar ordenado de menor a mayor.

El tamaño del problema es el tamaño del vector:

n= fin-ini+1

Para calcular la eficiencia de este algoritmo vamos a fijarnos en las 3 sentencias condicionales que tiene el programa, aunque sin dejarnos las dos operaciones elementales de inicialización y declaración de la variable "centro" que son de eficiencia O(1).

En el primer if, está el caso base en el que no encontramos el elemento, que es el peor caso en el problema de las búsquedas. Como la eficiencia de una sentencia condicional es el máximo entre la condición y el cuerpo, ambas son operaciones elementales, y por lo tanto O(1), asi que el máximo de esas dos órdenes es O(1).

En el segundo if vemos otro caso base, y es en el que encontramos el elemento y está en el centro de la parte del vector en la que estamos. Al ser una sentencia condicional, su eficiencia está definida por el máximo entre la condición y el cuerpo, su eficiencia es O(1) porque son operaciones elementas y el máximo de estas es O(1).

El tercer if es el caso en el que el elemento sea menor que el centro de la parte en la que nos encontramos. La eficiencia de esta sentencia viene definida por la llamada recursiva que hacemos en el cuerpo de esta, la cual es T(n/2), ya que llamamos a la mitad inferior del vector, es decir desde "ini" hasta uno antes de "centro".

Seguidamente vemos la última línea del código un return que hace otra llamada recursiva, pero esta vez con la mitad superior del vector, al revés que en el if anterior, asi que estaríamos en el mismo caso con una eficiencia de T(n/2).

En el mejor de los caso entramos en el segundo if, ya que elemento que queremos buscar está en el centro por lo tanto su eficiencia es de O(1).

En el peor caso haríamos llamadas recursivas hasta que no pudiéramos dividir más y entraríamos en el primer if. A estas llamadas siempre las llamaríamos con la mitad del vector que nos ha llegado, diciendo si el centro de esa parte es mayor que elemento lo llamaríamos con la mitad inferior a partir del centro de la parte del vector que tenemos, si por el contrario es menor, lo llamaríamos con la parte superior, lo opuesto. Gracias a que está ordenado podemos ir haciendo suposiciones de donde esta e ir quitándonos la mitad del tamaño del problema cada vez que hacemos una llamada.

Para resolver la recurrencia podemos elegir varios métodos, por ejemplo:

- *El caso base es O(1)*
- *El caso general será 1(condición del if)+ T(n/2).*

Cada vez que llamamos a la función recursiva se le llama con la mitad del vector por lo que desarrollándola tenemos:

$$1+T(n/2)= 2+T(n/4)= 3+T(n/8)= 4+T(n/16)=.....$$

Podemos ver que hay una pauta la cual es:

$$i+T(n/2^i)$$

Podemos resolver la ecuación habiendo un cambio de variables, sustituyendo la i como un $log_2(n)$:

$$log_2(n)+T(n/2^{log_2(n)})$$

Por las propiedades de los lo logaritmos, lo que hay dentro de la T es equivalente a 1. Por lo que nos queda que la eficiencia de la función recursiva es $O(log_2(n))$.

b)

```
void HeapSort(int *v, int n){

        double *apo=new double [n];
        int tamapo=0;

        for (int i=0; i<n; i++){
                insertar(apo,tamapo,v[i]);
        }
        for (int i=0; i<n; i++) {
                v[i]=apo[0];
                borrarRaiz(apo,tamapo);
        }
        delete [] apo;

}

void insertar(double *apo, int &tamapo, double valor){

        apo[tamapo]=valor;
        tamapo++;
        int aux =tamapo-1;
        bool fin =false;
        while (!fin) {
                int padre;
                if (aux==0) {
                        fin=true;
                }else{

                        if (aux%2==0) {
                                padre=(aux-2)/2;
                        }else{
                                padre=(aux-1)/2;
                        }

                        if (apo[padre] > apo[aux]) {
                                double tmp=apo[aux];
                                apo[aux]=apo[padre];
                                apo[padre]=tmp;
                                aux=padre;


                        }else{
                                fin=true;
                        }

                }
        }
}
```

```
void borrarRaiz(double *apo, int &tamapo){

  apo[0]=apo[tamapo-1];
  tamapo--;

  int aux=0;
  bool fin = false;

  while (!fin) {
    if (2*aux+1 >= tamapo) fin=true;
    else{
      int minhijo=2*aux+1;
      if ((minhijo+1 < tamapo) && (apo[minhijo]>apo[minhijo+1])) minhijo++;
      if (apo[aux]>apo[minhijo]) {
        double tmp = apo[aux];
        apo[aux]=apo[minhijo];
        apo[minhijo]=tmp;
        aux=minhijo;
      }else fin=true;
    }
  }

}
```

Este algoritmo resuelve el problema de la ordenación mediante la construcción de un APO, el cual va guardando el menor en su raíz, siendo este algoritmo no recursivo

El tamaño del problema será el tamaño del vector a ordenar n.

La eficiencia del algoritmo viene definida por 2 bucles for. Antes de los cuales hay dos líneas con sentencias simples, formadas por operaciones elementales con eficiencia O(1).

El primer for está formado por la llamada a una función. Para ver la eficiencia de este bucle primero vamos a ver la eficiencia de la función insertar.

La función insertar está formada por un bucle while. Antes del bucle while que engloba a varias sentencias condicionales, hay varias operaciones elementales todas ellas con eficiencia O(1). En el cuerpo del bucle while encontramos una operación elemental con eficiencia O(1), seguida de una sentencia condicional if-else. Su eficiencia está definida por la condición O(1), el cuerpo del if está definida por O(1) y el cuerpo del else, formado a su vez con dos condiciones if-else. La primera de esas dos sentencia if-else, tiene como eficiencia max{O(1),O(1),O(1)} que es O(1).La segunda de esas dos sentencias condiciones tiene como eficiencia max{O(1),O(1),O(1)} que es O(1). Podemos asumir entonces que else de la sentencia condicional if-else de dentro del while es O(1) y por lo tanto el cuerpo del while es O(1). Sabemos por teoría que while tiene una eficiencia de **O(g(n) + h(n)*(g(n)+f(n)))**. El bucle while dará n/2 vueltas, por lo que sabemos que la eficiencia del bucle será O($\log_2$(n)), por que como vemos si hacemos el else el padre siempre será la mitad de auxiliar menos uno o menos dos, y el auxiliar esta inicializado con el tamaño del vector, ósea n, asi que siempre ira de mitad en mitad.

Ahora sabemos que la eficiencia de la función insertar que es O($\log_2$(n)). El cuerpo del bucle for ejecuta n veces, es decir, tiene como eficiencia O(n* $\log_2$(n)).

En el segundo for encontramos la llamada a otra función, borrarRaiz, pero antes hay una operación elemental como eficiencia O(1). Ahora veamos la eficiencia de la función:

La función borrarRaiz está compuesta por varias operaciones elementales al principio de la función con eficiencia todas ellas O(1). Para calcular la eficiecnia del bucle while, vamos a calcular primero el cuerpo. Dentro del bucle while encontramos se sentencia condicional if-else, la eficiencia será la máxima entre la condición, O(1), el cuerpo del if, O(1) y el cuerpo del else, el cual está compuesto a su vez por dos sentencias condicionales, una sentencia if, que tiene como eficiencia max{O(1),O(1)}= O(1), y otra sentencia if-else, la cual tiene también como eficiencia el max{O(1),O(1),O(1)} que es O(1). Por lo que sabemos en teoría el while tiene una eficiencia de **O(g(n) + h(n)*(g(n)+f(n))).** El cuerpo del while se ejecuta n/2, ya que el proceso de borrar la raíz de un APO y mover los nodos no tarda más de $\log_2(n)$ veces. Es decir que el while tiene una eficiencia $O(\log_2(n))$ y la función también tendrá esa eficiencia ya que es la mayor de toda la función.

Continuando por el for, según la teoría tiene una eficiencia de **O(i(n)+g(n) + h(n)*(g(n)+f(n)+a(n))).** El cuerpo se ejecuta n veces, ya que recorre todos los elementos del vector, y tiene una eficiencia $O(\log_2(n))$. Por lo que la eficiencia del bucle for es $O(n^* O(\log_2(n)))$.

Por lo tanto la función tiene una eficiencia de $O(n^* O(\log_2(n)))$ que la eficacia mayor de las de la función en el peor de los casos. En el mejor de los casos sigue recorriendo lo mismo y haciendo los mismos pasos ya que dependen del tamaño, excepto cuando el vector este vacío, recorre todo el vector ya este ordenado o no. En este caso podemos decir que su orden es exacto.

Eficiencia Practica e Hibrida

Ahora veremos el tiempo práctico de los algoritmos HeapSort, MergeSort y burbuja.

Tiempos del MergeSort

n	Tiempo ejec.	Tiempo Teoric
10	8	1,32875974
20	7	3,45751256
40	13	8,51501127
70	27	17,1618439
80	24	20,2299948
100	43	53,8035251
200	76	61,1503204
400	179	138,300502
700	266	264,631621
800	316	308,600728
1000	398	398,627922
2000	878	877,255151
4000	1947	1914,50892
7000	3704	3576,44803
8000	2401	4149,01507
10000	2896	5315,03896
20000	6109	11430,071
40000	11994	24460,1281
70000	20515	45065,7985
80000	23799	52120,2286
100000	29865	66437,987
200000	62608	140875,905
400000	131216	297751,671
700000	239306	543671,166
800000	274816	627503,065
1000000	344095	797255,844
2000000	729341	1674511
4000000	1522758	3509020,61
7000000	2742897	6366843,48
8000000	3135772	7338038,44

Tiempos del HeapSort

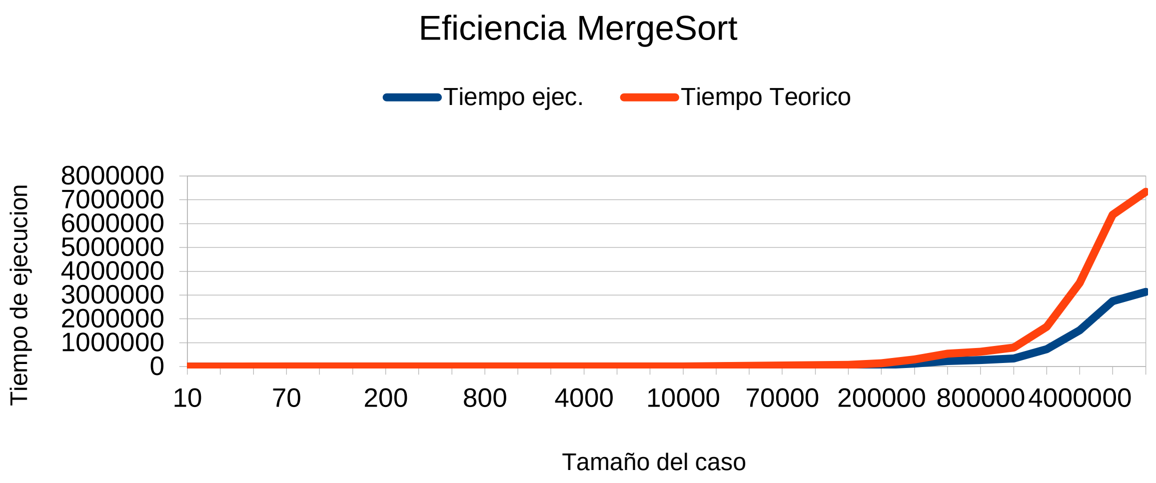
n	Tiempo ejec.	Tiempo Teoric
10	7	1,54640931
20	8	4,02384979
40	16	9,90976193
70	27	19,9729375
80	27	23,5436486
100	38	61,4897429
200	88	71,1666841
400	179	160,953992
700	333	307,978026
800	392	359,14923
1000	499	463,922792
2000	1099	1020,9487
4000	2388	2228,10364
7000	3247	4162,26678
8000	3531	4828,61975
10000	3584	6185,63723
20000	6548	13302,3056
40000	13469	28466,6736
70000	24776	52447,533
80000	28829	60657,472
100000	37068	77320,4654
200000	81465	163951,243
400000	178355	346523,109
700000	353475	632723,981
800000	425777	730287,464
1000000	551900	927845,585
2000000	1395212	1948794,29
4000000	3151598	4083794,81
7000000	6234052	7409726,33
8000000	7155275	8540002,09

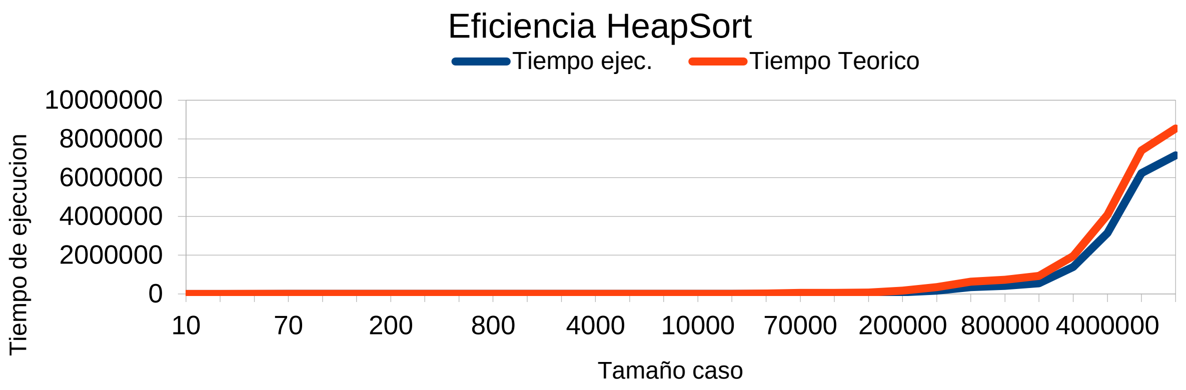
Tiempos burbuja

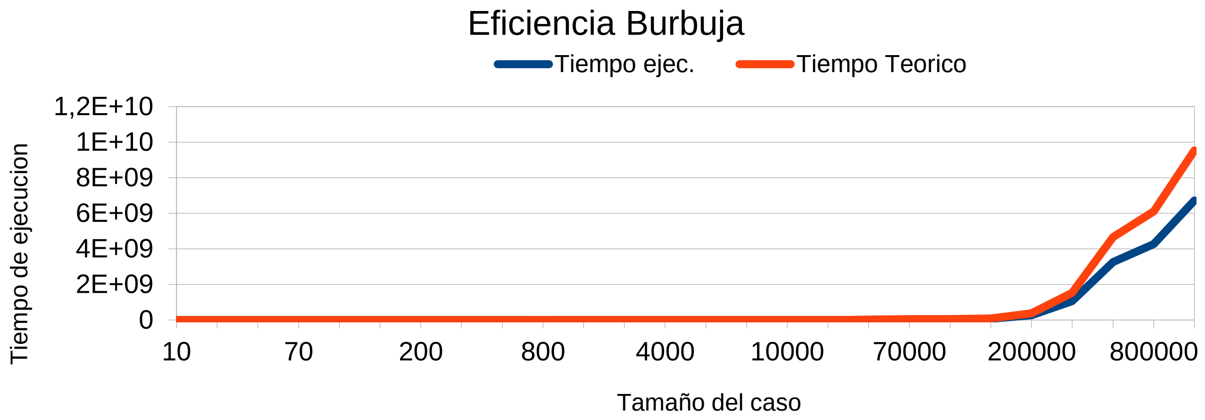
n	Tiempo ejec.	Tiempo Teori
10	2	0,95333998
20	7	3,81335992
40	25	15,2534397
70	76	46,713659
80	89	61,0137587
100	123	95,333998
200	542	381,335992
400	2419	1525,34397
700	5029	4671,3659
800	4950	6101,37587
1000	7230	9533,3998
2000	26185	38133,5992
4000	104293	152534,397
7000	317532	467136,59
8000	417634	610137,587
10000	657050	953339,98
20000	2655275	3813359,92
40000	10588822	15253439,7
70000	33051500	46713659
80000	42562523	61013758,7
100000	65864398	95333998
200000	263798546	381335992
400000	1055419472	1525343968
700000	3263796308	4671365902
800000	4273641395	6101375872
1000000	6721763583	9533399800

Como podemos apreciar en las tablas de arriba, el mergeSort es el más rápido de los otros dos algoritmos, aunque puede influir varios aspectos coando se mide el tiempo de ejcucion. Ambos algoritmo, tanto MergeSort como HeapSort, son $O(n* O(\log_2(n)))$. El algoritmo burbuja es con diferencia más lento.

Ahora veremos las gráficas de las funciones anteriores:







Las constantes ocultas en cada caso han sido

- HeapSort -> 0,046551558729003
- MergeSort -> 0,039999653862016
- Burbuja-> 0,0095334

Con las contantes ocultas junto con el orden de eficiencia en el peor caso, podemos acotar el tiempo de ejecución superiormente (k*f(n)).