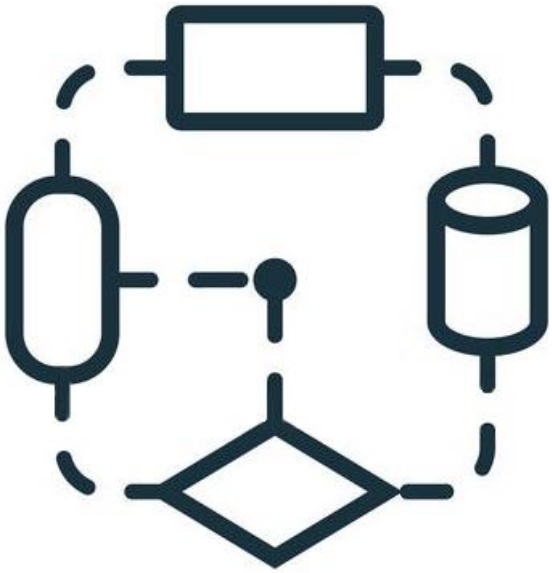
Análisis de Algoritmos

PRÁCTICA 1



Autor: Pablo Gómez Rivas

Materia: Lógica y Algorítmica

Grupo de Prácticas: GTA2

Fecha: Viernes, 1 de abril de 2022

Índice

Objetivo **3**

Resumen **3**

Sesión 01. Ordenar una matriz por filas **4**

Ejercicio 1: Tiempo medio obtenido por cada tamaño de entrada 4

Ejercicio 2: Gráfica tiempo de ejecución VS Tamaño de la entrada 4

Ejercicio 3: Tabla tomando logaritmos (en base 2) 5

Ejercicio 4: Gráfica comparativa tomando logaritmos 5

Ejercicio 5: Orden de complejidad del método experimentalmente 5

Ejercicio 6: Orden de complejidad del método teóricamente 6

Ejercicio 7: Estimar la constante c 7

Ejercicio 8: Tiempo de ejecución teórico para n = 100 000 7

Sesión 02. Resolver el Problema de la "Subsecuencia de suma máxima" **8**

Ejercicio 1: Estudio teórico de cada algoritmo 8

Ejercicio 2: Tiempo medio empleado VS Tamaño de entrada 11

Ejercicio 3: Gráfica del tiempo de ejecución de los 3 algoritmos 12

Ejercicio 4: Gráfica tomando logaritmos en base 2 12

Ejercicio 5: Tasas de crecimiento experimentales VS teóricas 13

Trabajo autónomo: BENCHMARKING **14**

¿Qué es un *benchmark* para computadores? 14

Principales campos de aplicación. 14

¿Qué tipo de pruebas realiza Linpack? ¿Qué es un MFLOP? 14

¿Qué es SPEC? ¿En qué campos desarrolla Benchmarks? 15

SPEC proporciona *benchmarks* para múltiples plataformas como puede ser en la nube como laaS (Infraestructure-as-a-Service), en CPUS, GPUs, tests de rendimientos a computadoras profesionales, dispositivos móviles, servidores de correo, dispositivos de almacenamiento, fuentes de alimentación, virtualización y servidores web. 16

Bibliografía **17**

Objetivo

El objetivo de esta práctica es implementar algoritmos en Java (IDE Eclipse) y realizar un análisis empírico de su tiempo de ejecución mediante pruebas de ejecución de los métodos, para recoger los datos obtenidos y posteriormente llegar a conclusiones sobre ellos.

Resumen

Queremos obtener el orden de complejidad del método matrizOrdenadaPorFilas() de forma experimental.

Clases que necesitamos implementar: paquete org.lya.sesion01

1. Clase MatrizEnterosCuadrada.java : Contiene la matriz de enteros, los getters y los siguientes métodos: campo privado con la matriz :
    a. int [][] matriz
    b. Dos constructores para crear la matriz:
        i. MatrizEnterosCuadrada(int numeroFilasCol): le asigna valores aleatorios entre 0 y 80.
        ii. MatrizEnterosCuadrada(int[][]m): a partir de otra matriz de entrada.
    c. Y los métodos:
        i. static void burbujaMejora(int[] array): ordenación por burbuja mejorado
        ii. MatrizEnterosCuadrada matrizOrdenadaPorFilas() : el método que queremos evaluar. Devuelve un nuevo objeto con la nueva matriz ordenada.
        iii. String toString() : método toString para pasar los elementos de la matriz a un String. Se recomienda revisar el fichero de test para comprobar qué resultado se debe obtener.
2. Clase TiemposMatrizEnterosCuadrada.java : con el main() y las pruebas para diferentes tamaños de matriz.

Sesión 01. Ordenar una matriz por filas

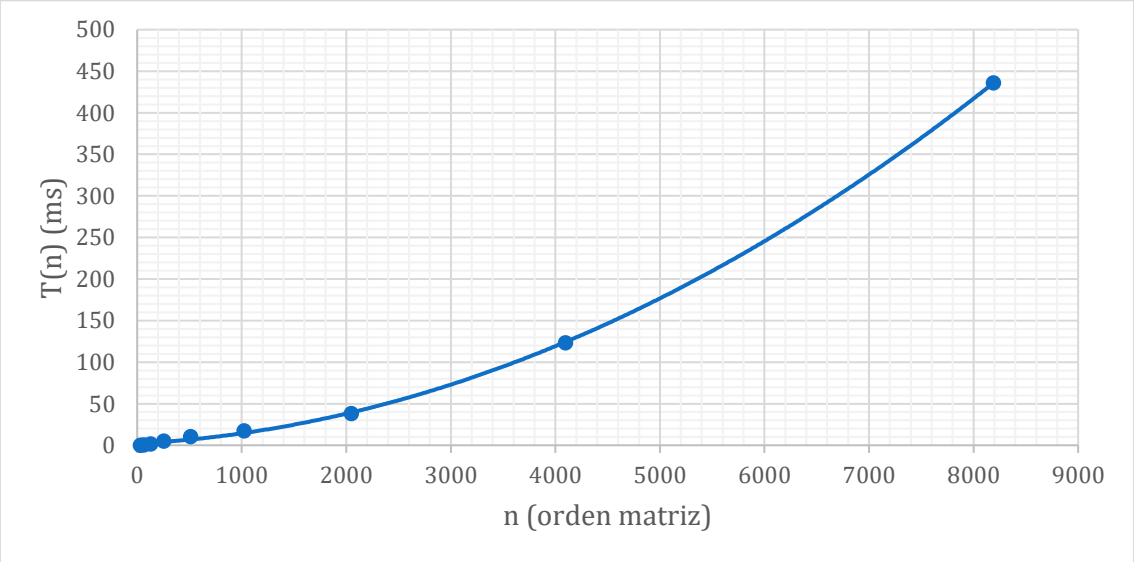
Ejercicio 1: Tiempo medio obtenido por cada tamaño de entrada

```
Matriz de dimensión n =   32     Tiempo medio = 188544,3333 nanosegundos
Matriz de dimensión n =   64     Tiempo medio = 625022,1111 nanosegundos
Matriz de dimensión n =  128     Tiempo medio =     1,6667 milisegundos
Matriz de dimensión n =  256     Tiempo medio =     5,2222 milisegundos
Matriz de dimensión n =  512     Tiempo medio =    10,4444 milisegundos
Matriz de dimensión n = 1024     Tiempo medio =    17,4444 milisegundos
Matriz de dimensión n = 2048     Tiempo medio =    38,2222 milisegundos
Matriz de dimensión n = 4096     Tiempo medio =   123,2222 milisegundos
Matriz de dimensión n = 8192     Tiempo medio =   436,1111 milisegundos
```

Tiempo de Ejecución VS Tamaño de la Entrada

n (orden matriz)	$\overline{T}$(n) (ms)
32	0,1885
64	0,625
128	1,6667
256	5,2222
512	10,4444
1024	17,4444
2048	38,2222
4096	123,2222
8192	436,1111

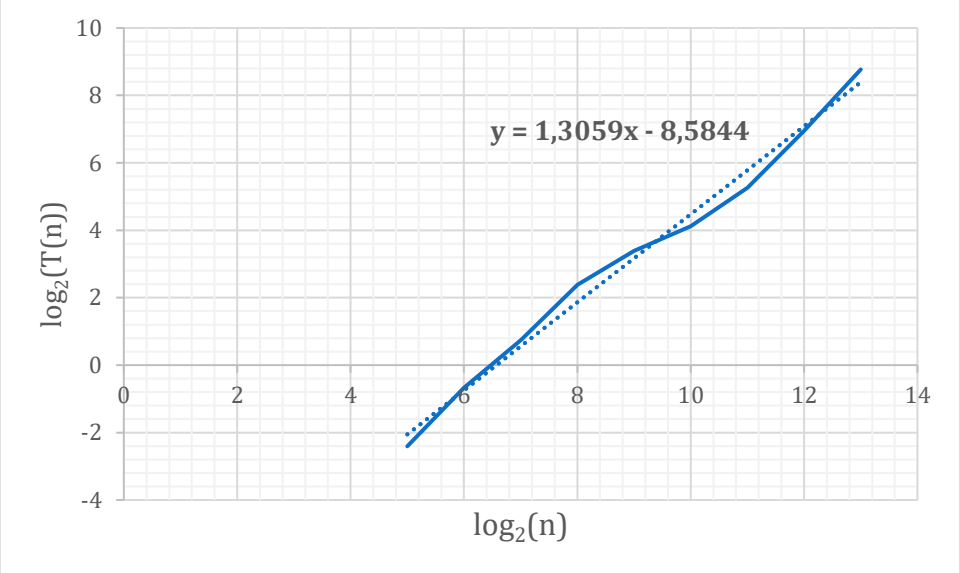
Ejercicio 2: Gráfica tiempo de ejecución VS Tamaño de la entrada



Ejercicio 3: Tabla tomando logaritmos (en base 2)

n (orden matriz)	$\overline{T}$(n) (ms)	$\log_2(n)$	$\log_2(\overline{T}(n))$
32	0,1885	**5**	-2,407363571
64	0,625	**6**	-0,678071905
128	1,6667	**7**	0,736994448
256	5,2222	**8**	2,384657711
512	10,4444	**9**	3,384657711
1024	17,4444	**10**	4,124692072
2048	38,2222	**11**	5,256338914
4096	123,2222	**12**	6,945118389
8192	436,1111	**13**	8,7685519

Ejercicio 4: Gráfica comparativa tomando logaritmos



Ejercicio 5: Orden de complejidad del método experimentalmente

Fijándonos en la anterior gráfica, podemos ver que la ecuación de la recta es:

y = 1,3059x - 8,5844

tiene como pendiente k= 1,3059. Por tanto, podemos decir que este algoritmo tiene orden de complejidad $O(n^{1,3059})$

Ejercicio 6: Orden de complejidad del método teóricamente

```
public MatrizEnterosCuadrada matrizOrdenadaPorFilas() {

        int[][] matrizNueva = new
int[matriz.length][matriz[0].length];

        for (int i = 0; i < matriz.length; i++) {

            for (int j = 0; j < matriz[0].length; j++)

                matrizNueva[i][j] = matriz[i][j];

            burbujaMejora(matrizNueva[i]);

        }

        return new MatrizEnterosCuadrada(matrizNueva);

    }
```

- El orden de complejidad de las líneas no marcadas es $O(1)$ ya que son operaciones elementales de tiempo constante.
- El orden de complejidad de los bucles FOR anidados `for (int i = 0; i < matriz.length; i++)` y `for (int j = 0; j < matriz[0].length; j++)` se calcula:

$$\sum_{i=0}^{n-1} \left(\sum_{j=0}^{n-1} d + P(i) \right)$$

donde c es la línea `matrizNueva[i][j] = matriz[i][j];` y P(i) es `burbujaMejora(matrizNueva[i]);`, cuyo orden de complejidad es $O(n^2)$.

Realizando las sumatorias queda:

$$\bar{T}(n) \leq n \cdot (n^2 + n \cdot d) \rightarrow \mathbf{\bar{T}(n) \leq n^3 + dn^2 \in O(n^3)}$$

Ejercicio 7: Estimar la constante c

Teniendo en cuenta que $\bar{T}(n) \leq c \cdot n^k, n = 8192\ y\ k = 1{,}3059$, podemos hallar el valor de c:

$$\bar{T}(8192) \leq c \cdot 8192^{1{,}3059}$$

$$436{,}1111 \leq c \cdot 8192^{1{,}3059}$$

$$c \geq 3{,}38144 \cdot 10^{-3}$$

Ejercicio 8: Tiempo de ejecución teórico para n = 100 000

Puesto que ya tenemos la expresión completa de $\bar{T}(n)$, bastaría con sustituir c en ella para n = 100 000.

$$\bar{T}(n) \leq 3{,}38144 \cdot 10^{-3} \cdot n^{1{,}3059}$$

$$\overline{\mathrm{T}}(100\ 000) \leq 3{,}38144 \cdot 10^{-3} \cdot 100\ 000^{1{,}3059}$$

$$\overline{\mathrm{T}}(100\ 000) \leq 11\ 444{,}6281 \text{ s}$$

Sesión 02. Resolver el Problema de la "Subsecuencia de suma máxima"

Ejercicio 1: Estudio teórico de cada algoritmo

- Algoritmo de Fuerza Bruta: examina mediante dos bucles anidados todas las posibles subsecuencias.
  - El primer bucle recorre todos los posibles comienzos de subsecuencia (para i = 0 hasta n= array.length -1).
  - El segundo bucle recorre todos los posibles finales de subsecuencia (para j = i hasta n = array.length -1).
  - El tercer bucle calcula la suma de la subsecuencia elegida, actualiza una solución mayor a la actual en caso de encontrarla, y actualiza las variables *primer* y *ultimo*.

```
public void SubsecuenciaFuerzaBruta() {

        for (int i = 0; i < array.length; i++) {

            for (int j = i; j < array.length; j++) {

                int sumaActual = 0;

                for (int k = i; k <= j; k++)

                    sumaActual += array[k];

                if (sumaActual > suma) {

                    suma = sumaActual;
                    primer = i;
                    ultimo = j;

                }
            }
        }
    }
```

$$\sum_{i=0}^{n-1} \left(\sum_{j=i}^{n-1} (c + \sum_{k=i}^{j} d) \right)$$

Desarrollando las sumatorias:

$$\bar{T}(n) \leq \frac{d}{6}(n^3 + 5n) + \frac{c}{2}(n^2 + n) \, \epsilon \, \boldsymbol{O(n^3)}$$

- Algoritmo de Fuerza Bruta: al eliminar un bucle anidado innecesario, reduciríamos el tiempo de ejecución del método a un orden menor, en este caso a $O \in (n^2)$.

```
public void SubsecuenciaMejorado() {

        for (int i = 0; i < array.length; i++) {

            int sumaActual = 0;

            for (int j = i; j < array.length; j++) {

                sumaActual += array[j];

                if (sumaActual > suma) {

                    suma = sumaActual;

                    primer = i;

                    ultimo = j;

                }
            }
        }
    }
```

$$\sum_{i=0}^{n-1} (c + \sum_{j=i}^{n-1} d)$$

Desarrollando las sumatorias:

$$\bar{T}(n) \leq \frac{d}{2}(n^2 + n) + cn \; \epsilon \; \boldsymbol{O(n^2)}$$

- Algoritmo Lineal: es el algoritmo más eficiente porque su mejora se salta un gran número de subsecuencias que nunca podrían ser la mejor, eliminando así otro bucle anidado. Pasa de probar cada subsecuencia a descartar aquellas que empiezan por un número negativo, saltándose el anterior bucle anidado e incrementar i.

```
public void SubsecuenciaLineal() {

        int sumaActual = 0;

        for (int i = 0, j = 0; j < array.length; j++) {

             sumaActual += array[j];

             if (sumaActual > suma) {

                  suma = sumaActual;

                  primer = i;

                  ultimo = j;

             } else if (sumaActual < 0) {

                  i = j + 1;

                  sumaActual = 0;

             }
        }
   }
```

$$\sum_{i=0}^{n-1} c$$

Desarrollando las sumatorias:

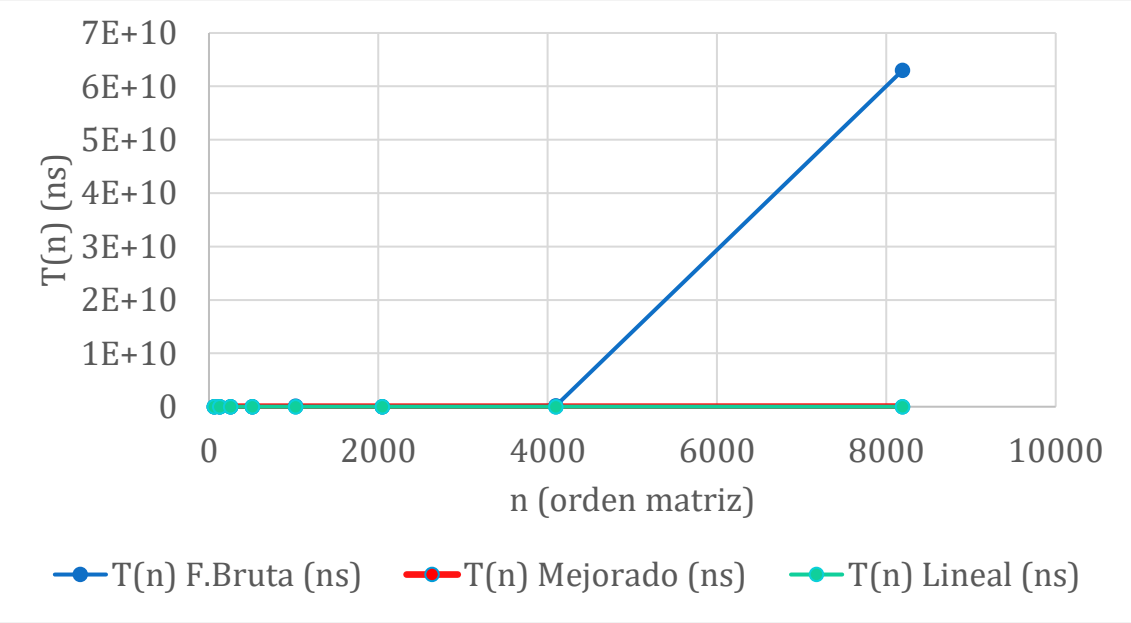
$$\bar{T}(n) \leq cn \; \epsilon \; \boldsymbol{O(n)}$$

Ejercicio 2: Tiempo medio empleado VS Tamaño de entrada

```
Dimension de la matriz, n = 64
Tiempo Medio Fuerza Bruta = 441688,6667 nanosegundos
Tiempo Medio Mejorado = 106266,6667 nanosegundos
Tiempo Medio Lineal = 4211,1111 nanosegundos

Dimension de la matriz, n = 128
Tiempo Medio Fuerza Bruta = 1023344,6667 nanosegundos
Tiempo Medio Mejorado = 323788,7778 nanosegundos
Tiempo Medio Lineal = 9955,5556 nanosegundos

Dimension de la matriz, n = 256
Tiempo Medio Fuerza Bruta = 2469677,5556 nanosegundos
Tiempo Medio Mejorado = 404710,8889 nanosegundos
Tiempo Medio Lineal = 19111,2222 nanosegundos

Dimension de la matriz, n = 512
Tiempo Medio Fuerza Bruta = 17622966,7778 nanosegundos
Tiempo Medio Mejorado = 217355,6667 nanosegundos
Tiempo Medio Lineal = 36933,3333 nanosegundos

Dimension de la matriz, n = 1024
Tiempo Medio Fuerza Bruta = 147722244,4444 nanosegundos
Tiempo Medio Mejorado = 504933,5556 nanosegundos
Tiempo Medio Lineal = 81722,1111 nanosegundos

Dimension de la matriz, n = 2048
Tiempo Medio Fuerza Bruta = 23730289,5556 nanosegundos
Tiempo Medio Mejorado = 760888,8889 nanosegundos
Tiempo Medio Lineal = 53844,2222 nanosegundos

Dimension de la matriz, n = 4096
Tiempo Medio Fuerza Bruta = 210670625,3333 nanosegundos
Tiempo Medio Mejorado = 3137210,8889 nanosegundos
Tiempo Medio Lineal = 106511,0000 nanosegundos

Dimension de la matriz, n = 8192
Tiempo Medio Fuerza Bruta = 63010884533.5556 nanosegundos
Tiempo Medio Mejorado = 26433044,7778 nanosegundos
Tiempo Medio Lineal = 424722,1111 nanosegundos
```

n (orden matriz)	$\overline{T}$(n) F.Bruta (ns)	$\overline{T}$(n) Mejorado (ns)	$\overline{T}$(n) Lineal (ns)
64	441688,6667	106266,6667	4211,1111
128	1023344,667	323788,7778	9955,5556
256	2469677,556	404710,8889	19111,2222
512	17622966,78	217355,6667	36933,3333
1024	147722244,4	504933,5556	81722,1111
2048	23730289,56	760888,8889	53844,2222
4096	210670625,3	3137210,889	106511
8192	63.010.884.534	26433044,78	424722,1111

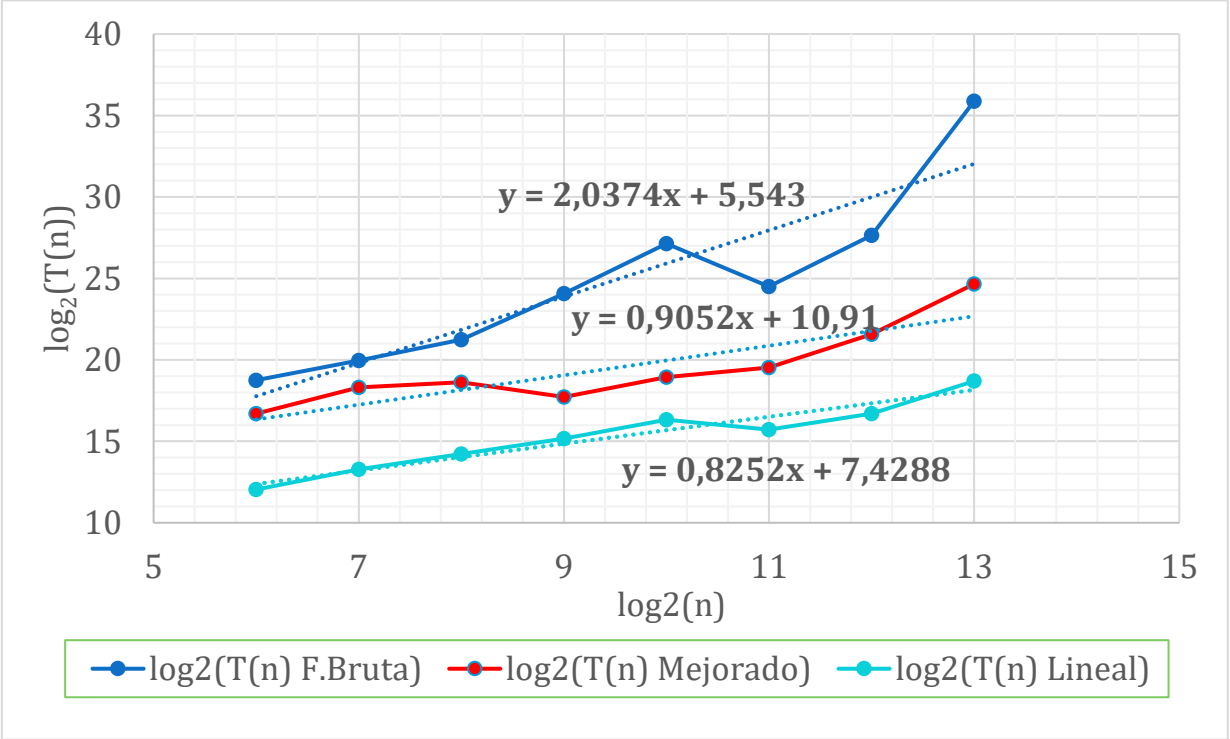
Ejercicio 3: Gráfica del tiempo de ejecución de los 3 algoritmos



Se puede observar como la gran diferencia entre el algoritmo de Fuerza bruta y los otros dos hace que estos queden en el eje X de la gráfica solapados.

Ejercicio 4: Gráfica tomando logaritmos en base 2

$\log_2$(n)	$\log_2(\overline{T}$(n) F.Bruta)	$\log_2(\overline{T}$(n) Mejorado)	$\log_2(\overline{T}$(n) Lineal)
6	18,75267029	16,6973296	12,03998522
7	19,9648607	18,30469346	13,28128612
8	21,23589126	18,62653214	14,22213242
9	24,07095348	17,72969818	15,17263585
10	27,13831185	18,94573403	16,31843885
11	24,50022637	19,53732627	15,71650392
12	27,65041393	21,58105108	16,70064291
13	35,87488201	24,65583928	18,69615969



La tasa de crecimiento de cada algoritmo la obtenemos fijándonos en la pendiente de cada una de sus rectas correspondientes:

- Algoritmo fuerza bruta: **y = 2,0374x + 5,543** → $O(n^{2,0374})$
- Algoritmo mejorado: **y = 0,9052x + 10,91** → $O(n^{0,9052})$
- Algoritmo lineal: **y = 0,8252x + 7,4288** → $O(n^{0,8252})$

Ejercicio 5: Tasas de crecimiento experimentales VS teóricas

Según el libro, las tasas de crecimiento teóricas para los algoritmos fuerza bruta, mejorado y lineal son respectivamente: $O(n^3)$, $O(n^2)$, $O(n)$.

Sin embargo, las que hemos obtenido experimentalmente no coinciden con estas, al igual que ocurre en la sesión 01, ya que no estamos trabajando con un tamaño de entrada lo suficientemente grande como para estimar de forma más precisa su orden de ejecución, puesto que nuestros computadores personales no lo permiten.

Trabajo autónomo: BENCHMARKING

¿Qué es un *benchmark* para computadores?

En informática, *benchmark* es el acto de ejecutar un programa de computadora, un conjunto de programas u otras operaciones, para evaluar el rendimiento relativo de un objeto, normalmente mediante la ejecución de una serie de pruebas y pruebas estándar en su contra. El término *benchmark* también se utiliza comúnmente para los propios programas de benchmarking elaborados.

La evaluación comparativa generalmente se asocia con la evaluación de las características de rendimiento del hardware de la computadora, por ejemplo, el rendimiento de operación de coma flotante de una CPU, pero hay circunstancias en las que la técnica también es aplicable al software. Los puntos de referencia de software se ejecutan, por ejemplo, contra compiladores o sistemas de gestión de bases de datos (DBMS).

Los *benchmarks* proporcionan un método para comparar el rendimiento de varios subsistemas en diferentes arquitecturas de chips/sistemas.

Principales campos de aplicación.

En el campo informático, los *benchmarks* se aplican para medir el rendimiento de un dispositivo o sus componentes y luego poder compararlos con otros. En este caso, es de gran utilidad a la hora de elegir que dispositivo o componente comprar o cuál va a ser usado para una determinada tarea.

También se suele usar en algunas áreas para estimar la potencia límite de un dispositivo o componente.

Además, podemos aplicar el benchmarking en el entorno empresarial.

Es el proceso a través del cual se hace seguimiento a otras empresas con el fin de evaluar sus productos, servicios, procesos u otros aspectos, compararlos con los propios y con los de otras empresas, identificar lo mejor, y adaptarlo a la propia empresa agregándoles mejoras.

¿Qué tipo de pruebas realiza Linpack? ¿Qué es un MFLOP?

El *benchmark* de LINPACK es una prueba de rendimiento, o benchmark, de uso muy difundido en la comunidad de cómputo de alto rendimiento. Este *benchmark* no pretende medir el desempeño general de un sistema. En vez de ello, evalúa el desempeño en un área muy específica: el cálculo de sistemas de ecuaciones lineales de alta densidad. Esta medida resulta útil para conocer las capacidades de equipo de cómputo de alto rendimiento, pues esta clase de equipos generalmente se utilizan para resolver ese tipo de problemas, de forma que el *benchmark* de LINPACK proporciona una idea bastante acertada del desempeño que el equipo tendrá en aplicaciones reales.

El uso del *benchmark* de LINPACK fue introducido en 1979 por Jack Dongarra, investigador de la Universidad de Tennessee en Knoxville, como parte del paquete LINPACK, un juego de bibliotecas matemáticas en FORTRAN para solución de sistemas de ecuaciones lineales. En general, el *benchmark* realiza la resolución de un sistema de ecuaciones generado aleatoriamente, expresado como una matriz de coeficientes que en la computadora están representados con números de punto flotante, normalmente a una precisión de 64 bits. El paso crucial de esta solución es la descomposición LU con pivoteo parcial de la matriz de coeficientes. Obtener la solución requiere $2/3n^3+2n^2$ operaciones de punto flotante, donde es la dimensión de la matriz. Normalmente se emplean valores de n=100 y n=1000.

Finalmente, el *benchmark* entrega un valor de rendimiento expresado en MFLOPS (millones de operaciones de punto flotante por segundo). Este valor es el que se suele emplear al hacer comparaciones entre diversos equipos.

LINPACK fue originalmente implementado en lenguaje FORTRAN para máquinas uniprocesador, vectoriales y SMP. A medida que los equipos MPP fueron cobrando auge, se hizo necesario el contar con una manera de comparar su desempeño, de forma que se desarrolló el *benchmark* HPL (High Performance LINPACK). HPL es una implementación del *benchmark* LINPACK escrita en lenguaje C, que puede ejecutarse en cualquier equipo que cuente con una implementación de MPI, lo cual incluye a prácticamente cualquier equipo MPP comercial y, desde luego, clusters tipo Beowulf.

HPL es el *benchmark* utilizado para medir el desempeño de las computadoras más rápidas del mundo, gracias a su portabilidad, su dependencia en otras bibliotecas que están ampliamente disponibles, particularmente MPI y BLAS4.7, y la capacidad de alterar fácilmente los parámetros de cálculo a fin de determinar la configuración óptima para obtener el mejor rendimiento.

El término FLOPS (Floating point operations per second) significa "operaciones de coma flotante por segundo", y es una unidad que se suele utilizar para medir los cálculos matemáticos que puede hacer por segundo una CPU y GPU. Un MFLOP o megaflop equivale a 10^6 FLOPS (millones de operaciones de punto flotante por segundo).

¿Qué es SPEC? ¿En qué campos desarrolla Benchmarks?

Standard Performance Evaluation Corporation (SPEC) es un consorcio sin fines de lucro que incluye a vendedores de computadoras, integradores de sistemas, universidades, grupos de investigación, publicadores y consultores de todo el mundo. Tiene dos objetivos: crear un *benchmark* estándar para medir el rendimiento de computadoras y controlar y publicar los resultados de estos tests.

SPEC proporciona *benchmarks* para múltiples plataformas como puede ser en la nube como laaS (Infraestructure-as-a-Service), en CPUS, GPUs, tests de rendimientos a computadoras profesionales, dispositivos móviles, servidores de correo, dispositivos de almacenamiento, fuentes de alimentación, virtualización y servidores web.

- Recoge en una tabla las características de los 10 primeros supercomputadores de la TOP500 list de noviembre de 2021. Indica también para qué se usan esos supercomputadores y cuánta energía consumen.

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	Supercomputer Fugaku	7,630,848	442,010.0	537,212.0	29,899
2	Summit	2,414,592	148,600.0	200,794.9	10,096
3	Sierra	1,572,480	94,640.0	125,712.0	7,438
4	Sunway TaihuLight	10,649,600	93,014.6	125,435.9	15,371
5	Perlmutter	761,856	70,870.0	93,750.0	2,589
6	Selene	555,52	63,460.0	79,215.0	2,646
7	Tianhe-2A	4,981,760	61,444.5	100,678.7	18,482
8	JUWELS Booster Module	449,28	44,120.0	70,980.0	1,764
9	HPC5	669,76	35,450.0	51,720.8	2,252
10	Voyager-EUS2	253,44	30,050.0	39,531.2	

Bibliografía

- Apuntes de la asignatura
- Benchmark (computing) - Wikipedia
- Qué son los teraflops y qué miden exactamente – Xataka
- Standard Performance Evaluation Corporation - Wikipedia
- November 2021 | TOP500