Algoritmos - Asier Llano - AG1

July 9, 2023

Actividad Guiada 1

Asier Llano

Link repostorio github: [https://github.com/asierllano/03MIAR—Algoritmos-de-Optimizacion—2023]

1 Torres de Hanoi

Esta aplicación consiste en hacer una aplicación que solucione el problema de las torres de Hanoi. He hecho la solución muy similar a la presentada en la clase, con las siguientes mejoras menores: * En vez de limitar para N=1 he limitado para N=0 para no tener que realizar el movimiento para N=1 también * No he pasado el pivote porque hay que tener en cuenta que origen + destino + pivote = 6 siempre. * En vez de imprimir texto, he creado un generador en python, para que el resultado sea iterable y así se puede utilizar los movimientos para cualquier aplicación, no solamente escribir el texto. * Incluidos parámetros por defecto, porque tradicionalmente el problema de las torres de Hanoi empieza en la primera columna y termina en la última.

[ ]:
```python
# Función que devuelve un iterador con los movimientos
# que soluciona las torres de Hanoi
def torres_hanoi(N, origen=1, destino=3):
    if N > 0:
        pivote = 6 - origen - destino
        yield from torres_hanoi(N-1, origen, pivote)
        yield (origen, destino)
        yield from torres_hanoi(N-1, pivote, destino)
```

Realizamos dos aplicaciones diferentes, una que muestra los movimientos para N=4 y otra que cuenta los movimientos necesarios para diferentes valores de N

[ ]:
```python
# Pinta los movimientos para N=4
print("MOVIMIENTOS PARA N=4")
movimiento = 1
for (origen, destino) in torres_hanoi(4):
    print(f"{movimiento}. Muevo disco de {origen} a {destino}")
    movimiento+=1

# Comprueba el número de movimientos para diferentes valores de N
```

```python
print()
print("NUMERO DE MOVIMIENTOS")
for N in range(1,21):
    movimientos = len(list(torres_hanoi(N)))
    print(f"N={N} requiere {movimientos} movimientos")
```

```
MOVIMIENTOS PARA N=4
1. Muevo disco de 1 a 2
2. Muevo disco de 1 a 3
3. Muevo disco de 2 a 3
4. Muevo disco de 1 a 2
5. Muevo disco de 3 a 1
6. Muevo disco de 3 a 2
7. Muevo disco de 1 a 2
8. Muevo disco de 1 a 3
9. Muevo disco de 2 a 3
10. Muevo disco de 2 a 1
11. Muevo disco de 3 a 1
12. Muevo disco de 2 a 3
13. Muevo disco de 1 a 2
14. Muevo disco de 1 a 3
15. Muevo disco de 2 a 3

NUMERO DE MOVIMIENTOS
N=1 requiere 1 movimientos
N=2 requiere 3 movimientos
N=3 requiere 7 movimientos
N=4 requiere 15 movimientos
N=5 requiere 31 movimientos
N=6 requiere 63 movimientos
N=7 requiere 127 movimientos
N=8 requiere 255 movimientos
N=9 requiere 511 movimientos
N=10 requiere 1023 movimientos
N=11 requiere 2047 movimientos
N=12 requiere 4095 movimientos
N=13 requiere 8191 movimientos
N=14 requiere 16383 movimientos
N=15 requiere 32767 movimientos
N=16 requiere 65535 movimientos
N=17 requiere 131071 movimientos
N=18 requiere 262143 movimientos
N=19 requiere 524287 movimientos
N=20 requiere 1048575 movimientos
```

2 Devolución de cambio

Esta aplicación realiza la devolución de cambio en base a un sistema monetario. Es un código muy similar al realizado en clase con las siguientes mejoras: * Permite sistemas monetarios no ordenados * Utiliza división de enteros, en vez de división en coma flotante y después extraer la parte entera. * En vez de pintar el resultado es un generador iterable con las monedas y su cantidad para poder utilizarlo en cualquier tipo de aplicaciones. * En vez de sumar la cantidad hasta llegar al total, lo va restando del total, lo que simplifica el cálculo. * Proporciona una excepción si no se ha podido devolver el cambio

[ ]:
```
# Función que devuelve el cambio de moneda basado en un sistema monetario
def cambio_moneda(total, sistema):
    for moneda in sorted(sistema, reverse = True):
        cantidad = total // moneda
        if cantidad != 0:
            yield (moneda, cantidad)
        total -= cantidad * moneda
    if total != 0:
        raise ValueError("No es posible devolver este cambio")
```

[ ]:
```
# Aplicación que devuelve el cambio de 54
print("El cambio de 54 es:")
for (moneda, cantidad) in cambio_moneda(54, [25, 1, 10, 5]):
    print(f" * {cantidad} monedas de {moneda}")
```

```
El cambio de 54 es:
 * 2 monedas de 25
 * 4 monedas de 1
```

3 Encontrar los dos puntos más cercanos

Dado un conjunto de puntos devolver los puntos más cercanos en 1D, 2D, 3D… Empezar utilizando fuerza bruta y optimizarlo de forma recursiva

3.1 Implementación por fuerza bruta

La implementación por fuerza bruta requiere probar todas las parejas de números diferentes.

Hay que tener en cuenta que para n puntos, hay un total de $\frac{n \cdot (n-1)}{2}$ parejas. Por eso el algoritmo es $O(n^2)$

[ ]:
```
import random
import math

# Genero una lista de puntos aleatoria
def puntos_aleatorios(number, dimensions, maxvalue=10000):
    return [tuple(random.randrange(1,maxvalue) for d in range(dimensions)) \
            for i in range(number)]
```

```python
#  Dado un conjunto de puntos calcula los más cercanos por fuerza bruta
def puntos_cercanos_fuerza_bruta(puntos):
    if len(puntos) < 2:
        raise ValueError("No se puede calcular la distancia entre puntos si no hay 2")
    d_mejor = float("inf")
    resultado = (None, None)
    range_k = range(len(puntos[0]))
    for i in range(len(puntos)):
        p1 = puntos[i]
        for p2 in puntos[:i]:
            d = 0
            for k in range_k:
                dk = p1[k] - p2[k]
                d += dk*dk
                # Comprabmos esto en cada dimensión en vez de al final
                # para terminar prematuramente si la distancia ya
                # es mala sin computar todas (lo será habitualmente)
                if d_mejor <= d:
                    break
            else:
                d_mejor = d
                resultado = (p1, p2)
    return resultado
```

Probamos el resultado para 3D con 5 elementos:

[ ]:
```python
puntos=puntos_aleatorios(5,3)
cercanos=puntos_cercanos_fuerza_bruta(puntos)
print(f"Puntos más cercanos: {cercanos} en:")
for p in puntos:
    print(f" * {p}")
```

```
Puntos más cercanos: ((2266, 1832, 3336), (6176, 2896, 1670)) en:
 * (378, 8190, 1457)
 * (6176, 2896, 1670)
 * (2266, 1832, 3336)
 * (6390, 7802, 1241)
 * (4100, 4589, 7110)
```

3.2 Implementación recursiva

Después de bucear en la bibliografía el mejor algoritmo recursivo que he encontrado (y entendido) es el publicado por Khuller and Matias en 1995: https://www.cs.umd.edu/~samir/grant/cp.pdf

Lo he encontrado por la entrada de la Wikipedia en inglés de este problema: https://en.wikipedia.org/wiki/Closest_pair_of_points_problem#Linear-time_randomized_algorithms

Este algoritmo está muy bien explicado en el artículo y muy bien resumido en la entrada de la Wikipedia. Además, tiene la particularidad de funcionar en cualquier número de dimensiones.

Los pasos son para encontrar la distancia mínima entre dos puntos de un conjunto S:

0. Inicializar S_1 para que sea S e $i = 1$
1. Elegir un punto p_i al azar de la colección S_i, computar la distancia entre p_i y todo el resto de los puntos de S_i. Llamaremos d_i a esa distancia mínima.
2. Construir una malla de paso $\frac{d}{2\sqrt{k}}$ (esta es la generalización para el caso N-dimensional donde k es el número de dimensiones). Crear un $S_{i+1} = S_i - X_i$ donde X_i es el conjunto de puntos que no tienen vecinos en una vecindad de Moore.
3. Si $S_{i+1} = \emptyset$ entonces $i = i + 1$ e ir al paso 1, sino entonces continuar al paso 4.
4. Construir un mallado de S (el original), de paso d_i. Para cada punto calcular la distancia con sus vecinos y elegir el mínimo.

La complejidad del algoritmo es $O(n)$ por el hecho de que estadísticamente en el paso 2 se eliminan la mayoría de puntos y la colección decrece geométricamente, por lo que el tiempo que domina es el de la primera iteración.

Se recomienda ver el artículo para más detalle.

[ ]:
```python
# Infinitio utilizado como distancia máxima
infinito = float("inf")

# Dada una colección de puntos devuelve el punto más cercano a uno dado
#   p:        Punto para calcular su más cercano
#   puntos:   Lista de pntos para obtener el mas cercano
# devuelve: (p2, distancia)
#   Tupla con el punto p2 y la distancia, que son los más cercanos a p.
def punto_mas_cercano(p, puntos):
    min_distancia = infinito
    min_punto = None
    range_k = range(len(p))
    for punto in puntos:
        if p is punto:
            continue
        d = 0
        for k in range_k:
            pk = p[k]
            puntok = punto[k]
            dk = pk - puntok
            d += dk*dk
            if d >= min_distancia:
                break
        else:
            min_distancia = d
            min_punto = punto
    return (min_punto, math.sqrt(min_distancia))
```

```python
# Dado un conjunto de puntos encasillarlos en una malla jerárquica
# dado un intervalo
#  puntos: Lista de puntos
#  inrtervalo: Intervalo entre los puntos
# Devuleve: Una malla de jerárquica
def puntos_a_malla(puntos, intervalo):
    malla = {}
    for p in puntos:
        m = malla
        for pi in p[:-1]:
            pci = int(pi // intervalo)
            if pci in m:
                m = m[pci]
            else:
                newm = {}
                m[pci] = newm
                m = newm
        pci = int(p[-1] // intervalo)
        if pci in m:
            m[pci].append(p)
        else:
            m[pci] = [p]
    return malla

# Dada una malla jerárquica devuelve los puntos que tienen algún vecino
#   malla: Malla generada con puntos_a_malla
#   alrededor: Variable intermedia para hacer recursividad de los puntos
#              de alrededor
def vecinos_filtro(malla, alrededor=[]):
    # Va por todas las claves de la malla central
    resultado=[]
    for (c,subm) in malla.items():
        # Si estamos en la última dimensión
        if type(subm) is list:
            if len(subm) > 1 or (c+1) in malla or (c-1) in malla:
                resultado += subm
            else:
                for a in alrededor:
                    if (c in a) or ((c+1) in a) or ((c-1) in a):
                        resultado += subm
                        break
        # Sino hacemos recursivo par la siguiente dimension
        else:
            suba=[]
            for a in alrededor:
                if c in a:
                    suba.append(a[c])
```

```
                if (c+1) in a:
                    suba.append(a[c+1])
                if (c-1) in a:
                    suba.append(a[c-1])
            if (c+1) in malla:
                suba.append(malla[c+1])
            if (c-1) in malla:
                suba.append(malla[c-1])
            resultado += vecinos_filtro(subm, suba)
    return resultado

# Generador que devuelve tuplas con puntos y sus vecinos
#   malla: Malla generada con puntos_a_malla
# genera: tuplas con listas de puntos y su lista de vecinos
def vecinos_lista(malla, alrededor=[]):
    # Va por todas las claves de la malla central
    for (c,subm) in malla.items():
        # Si estamos en la última dimensión
        if type(subm) is list:
            vecinos=[]
            vecinos+=subm
            if c+1 in malla:
                vecinos += malla[c+1]
            if c-1 in malla:
                vecinos += malla[c-1]
            for a in alrededor:
                if c in a:
                    vecinos += a[c]
                if c+1 in a:
                    vecinos += a[c+1]
                if c-1 in a:
                    vecinos += a[c-1]
            yield (subm,vecinos)
        # Sino hacemos recursivo par la siguiente dimension
        else:
            suba=[]
            for a in alrededor:
                if c in a:
                    suba.append(a[c])
                if (c+1) in a:
                    suba.append(a[c+1])
                if (c-1) in a:
                    suba.append(a[c-1])
            if (c+1) in malla:
                suba.append(malla[c+1])
            if (c-1) in malla:
                suba.append(malla[c-1])
```

```
            yield from vecinos_lista(subm, suba)

# Consigue la distancia mínima usando el algoritmo de Khuller and Matias 1995
#   puntos: Lista de puntos, cada punto siendo una tupla de la dimensión deseada
# devuelve: (p1, p2)
#   tupla con los puntos
def puntos_cercanos(puntos):
    # El algoritmos se plantea recursivo, eliminando puntos y repitiendo
    # con el resultado, pero con un bucle lo convertimos en iterativo
    s = puntos
    while True:
        # Paso 1: Elige un punto al azar y compunta la mínima distancia a todo
        #         el conjunto
        randindex = random.randint(0,len(s)-1)
        p1 = s[randindex]
        (p2, d) = punto_mas_cercano(p1, s)
        # Si la distancia ya es nula, entonces eso no se puede mejorar
        if math.isclose(d, 0):
            return (p1, p2)
        # Paso 2: Realizamos un mallado y elimina los que no tienen vecinos
        intervalo = d/(2*math.sqrt(len(p1)))
        malla = puntos_a_malla(s, intervalo)
        filtrado = vecinos_filtro(malla)
        # Paso 3: Condición de bucle y parada
        if len(filtrado) == 0:
            break
        s = filtrado
    # Paso 4:
    # Primerlo prefiltramos los que tienen vecinos con nuestra malla
    # jerárquica optimizada que es más rápido que la lista de vecinos
    # y después hacemos la lista de vecinos.
    intervalo = d
    malla = puntos_a_malla(puntos, intervalo)
    puntos = vecinos_filtro(malla)
    malla = puntos_a_malla(puntos, intervalo)
    for (puntos, vecinos) in vecinos_lista(malla):
        for p in puntos:
            (vp, vd) = punto_mas_cercano(p, vecinos)
            if vd < d:
                p1 = p
                p2 = vp
                d = vd
    # Devuelve el resultado
    return (p1, p2)
```

3.2.1 Coprobación del correcto funcionamiento del algoritmo

Comprobamos con 100 ejemplos de un número aleatorio de puntos (entre 2 y 1000), de un número aleatorio de dimensiones (entre 1 y 10) que la distancia por fuerza bruta y la distancia por el algoritmo Khuler-and-Matias dan el mismo resultado.

[ ]:
```
# Calcula la distancia entre puntos
#   a: Primer punto para calcular la distancia
#   b: Segundo punto para calcular la distancia
def distancia(a,b):
    result = 0
    for i in range(len(a)):
        d = a[i] - b[i]
        result += d*d
    return math.sqrt(result)

# Pone a prueba el algoritmo Khuler-and-Matias comparando su resultado con
# el de fuerza bruta, 100 veces, en un número aleatorio de puntos entre 2
# y 1000, entre 1 y 10 dimensiones.
for i in range(100):
    puntos=puntos_aleatorios(random.randint(2,1000),random.randint(1,10))
    cercanos=puntos_cercanos_fuerza_bruta(puntos)
    cercanos2=puntos_cercanos(puntos)
    d=distancia(cercanos[0], cercanos[1])
    d2=distancia(cercanos2[0], cercanos2[1])
    assert(math.isclose(d,d2))
```

3.2.2 Comprobación del rendimiento del algoritmo

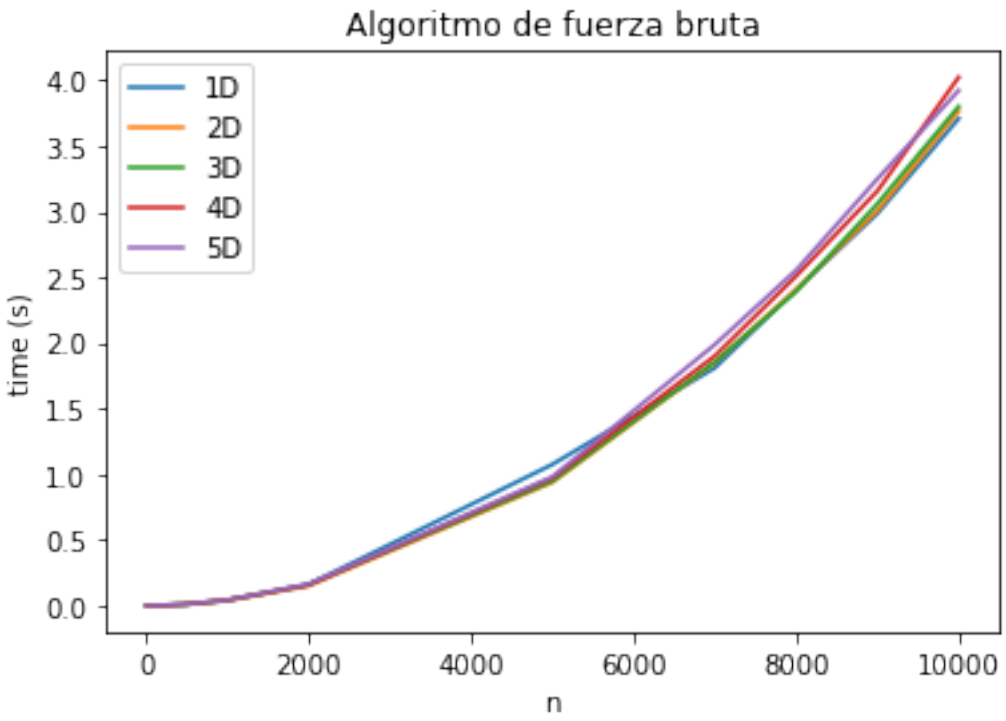
Vamos a hacer una gráfica del tiempo que tarda en ejecutarse el algoritmo en funciónd el número de nodos tanto para el caso de fuerza bruta como para el caso del algoritmo Khuler-and-Matias. Así comprobaremos que uno es $O(n^2)$ y el otro es $O(n)$.

[ ]:
```
import timeit
import matplotlib.pyplot as plt
x = [10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 7000, 8000, 9000, 10000]
y = {}
for dim in [1, 2,3,4,5]:
    print(f"Dimensiones: {dim}D")
    ydim = []
    for num in x:
        sets = 10 if num < 3000 else 1
        reps = 5000*5000//(sets*num*num)+1
        t_avg = 0
        for r in range(sets):
            puntos=puntos_aleatorios(num,dim)
            t = timeit.timeit(lambda:puntos_cercanos_fuerza_bruta(puntos),\
                              number=reps)
```

```
            t_avg += t
        t_avg /= sets*reps
        ydim.append(t_avg)
        k = t_avg / (num*num)
        print(f"  puntos: {num} tiempo: {t_avg} s ({k}*n^2 s)")
    y[dim] = ydim
plt.plot(x,y[1],x,y[2],x,y[3],x,y[4],x,y[5])
plt.xlabel("n")
plt.ylabel("time (s)")
plt.title('Algoritmo de fuerza bruta')
plt.legend(["1D", "2D", "3D", "4D", "5D"])
plt.show()
```

```
Dimensiones: 1D
  puntos: 10 tiempo: 4.3941142756761155e-06 s (4.3941142756761154e-08*n^2 s)
  puntos: 20 tiempo: 1.5766435274430433e-05 s (3.941608818607608e-08*n^2 s)
  puntos: 50 tiempo: 9.485385194606893e-05 s (3.794154077842757e-08*n^2 s)
  puntos: 100 tiempo: 0.00037838214462478293 s (3.7838214462478295e-08*n^2 s)
  puntos: 200 tiempo: 0.001478447461970121 s (3.6961186549253024e-08*n^2 s)
  puntos: 500 tiempo: 0.009025464654868384 s (3.610185861947354e-08*n^2 s)
  puntos: 1000 tiempo: 0.03906696763103052 s (3.906696763103052e-08*n^2 s)
  puntos: 2000 tiempo: 0.15852804880414623 s (3.963201220103656e-08*n^2 s)
  puntos: 5000 tiempo: 1.0738432399957674 s (4.295372959830695e-08*n^2 s)
  puntos: 7000 tiempo: 1.8099763330246788 s (3.693829251070773e-08*n^2 s)
  puntos: 8000 tiempo: 2.406136620993493 s (3.759588470302333e-08*n^2 s)
  puntos: 9000 tiempo: 2.985845058981795 s (3.686228467878759e-08*n^2 s)
  puntos: 10000 tiempo: 3.70863192298566 s (3.70863192298566e-08*n^2 s)
Dimensiones: 2D
  puntos: 10 tiempo: 5.082779332916181e-06 s (5.0827793329161815e-08*n^2 s)
  puntos: 20 tiempo: 1.799401732521372e-05 s (4.49850433130343e-08*n^2 s)
  puntos: 50 tiempo: 0.00010172478881056098 s (4.0689915524224395e-08*n^2 s)
  puntos: 100 tiempo: 0.00039425715019108273 s (3.942571501910827e-08*n^2 s)
  puntos: 200 tiempo: 0.0015645332539284099 s (3.9113331348210246e-08*n^2 s)
  puntos: 500 tiempo: 0.009310118200797165 s (3.7240472803188655e-08*n^2 s)
  puntos: 1000 tiempo: 0.037675241234440666 s (3.7675241234440666e-08*n^2 s)
  puntos: 2000 tiempo: 0.1487915156991221 s (3.719787892478053e-08*n^2 s)
  puntos: 5000 tiempo: 0.9373322689934867 s (3.749329075973947e-08*n^2 s)
  puntos: 7000 tiempo: 1.8498225099756382 s (3.775147979542119e-08*n^2 s)
  puntos: 8000 tiempo: 2.406293777981773 s (3.75983402809652e-08*n^2 s)
  puntos: 9000 tiempo: 3.008845361007843 s (3.714623902478819e-08*n^2 s)
  puntos: 10000 tiempo: 3.7649581790028606 s (3.764958179002861e-08*n^2 s)
Dimensiones: 3D
  puntos: 10 tiempo: 6.202575157094275e-06 s (6.202575157094274e-08*n^2 s)
  puntos: 20 tiempo: 2.0228686147073556e-05 s (5.057171536768389e-08*n^2 s)
  puntos: 50 tiempo: 0.00011131160918539009 s (4.4524643674156036e-08*n^2 s)
  puntos: 100 tiempo: 0.00041106703265178653 s (4.110670326517865e-08*n^2 s)
  puntos: 200 tiempo: 0.0016920057269984798 s (4.230014317496199e-08*n^2 s)
```

```
  puntos: 500 tiempo: 0.009789650936198251 s (3.9158603744793e-08*n^2 s)
  puntos: 1000 tiempo: 0.03938790453248657 s (3.938790453248657e-08*n^2 s)
  puntos: 2000 tiempo: 0.15771853189799004 s (3.942963297449751e-08*n^2 s)
  puntos: 5000 tiempo: 0.9472622924949974 s (3.78904916997999e-08*n^2 s)
  puntos: 7000 tiempo: 1.85457903999486 s (3.78485518366298e-08*n^2 s)
  puntos: 8000 tiempo: 2.3909706140111666 s (3.7358915843924475e-08*n^2 s)
  puntos: 9000 tiempo: 3.059303418995114 s (3.7769178012285355e-08*n^2 s)
  puntos: 10000 tiempo: 3.8004866859992035 s (3.8004866859992034e-08*n^2 s)
Dimensiones: 4D
  puntos: 10 tiempo: 7.076614375321116e-06 s (7.076614375321116e-08*n^2 s)
  puntos: 20 tiempo: 2.31907960326214e-05 s (5.7976990081553504e-08*n^2 s)
  puntos: 50 tiempo: 0.00012099717292280125 s (4.83988691691205e-08*n^2 s)
  puntos: 100 tiempo: 0.00046650890877333087 s (4.6650890877333086e-08*n^2 s)
  puntos: 200 tiempo: 0.0017331372984381954 s (4.332843246095489e-08*n^2 s)
  puntos: 500 tiempo: 0.01016922328192529 s (4.067689312770116e-08*n^2 s)
  puntos: 1000 tiempo: 0.04113200550103405 s (4.113200550103405e-08*n^2 s)
  puntos: 2000 tiempo: 0.15608099330565892 s (3.902024832641473e-08*n^2 s)
  puntos: 5000 tiempo: 0.9699472754873568 s (3.8797891019494274e-08*n^2 s)
  puntos: 7000 tiempo: 1.9002478410257027 s (3.878056818419801e-08*n^2 s)
  puntos: 8000 tiempo: 2.5096150969911832 s (3.921273589048724e-08*n^2 s)
  puntos: 9000 tiempo: 3.1572354310192168 s (3.897821519776811e-08*n^2 s)
  puntos: 10000 tiempo: 4.023596863989951 s (4.023596863989951e-08*n^2 s)
Dimensiones: 5D
  puntos: 10 tiempo: 8.278020199145834e-06 s (8.278020199145834e-08*n^2 s)
  puntos: 20 tiempo: 2.8427797729185068e-05 s (7.106949432296267e-08*n^2 s)
  puntos: 50 tiempo: 0.00014857581088465493 s (5.943032435386197e-08*n^2 s)
  puntos: 100 tiempo: 0.0005297991493763217 s (5.297991493763217e-08*n^2 s)
  puntos: 200 tiempo: 0.0018811433254057423 s (4.702858313514356e-08*n^2 s)
  puntos: 500 tiempo: 0.010739359591124495 s (4.2957438364497983e-08*n^2 s)
  puntos: 1000 tiempo: 0.04154000703323012 s (4.154000703323011e-08*n^2 s)
  puntos: 2000 tiempo: 0.16233125340368132 s (4.058281335092033e-08*n^2 s)
  puntos: 5000 tiempo: 0.9791736214974662 s (3.916694485989865e-08*n^2 s)
  puntos: 7000 tiempo: 1.9870085760194343 s (4.055119542896805e-08*n^2 s)
  puntos: 8000 tiempo: 2.5533993099816144 s (3.989686421846273e-08*n^2 s)
  puntos: 9000 tiempo: 3.248801268026 s (4.010865762995062e-08*n^2 s)
  puntos: 10000 tiempo: 3.9233242100162897 s (3.92332421001629e-08*n^2 s)
```



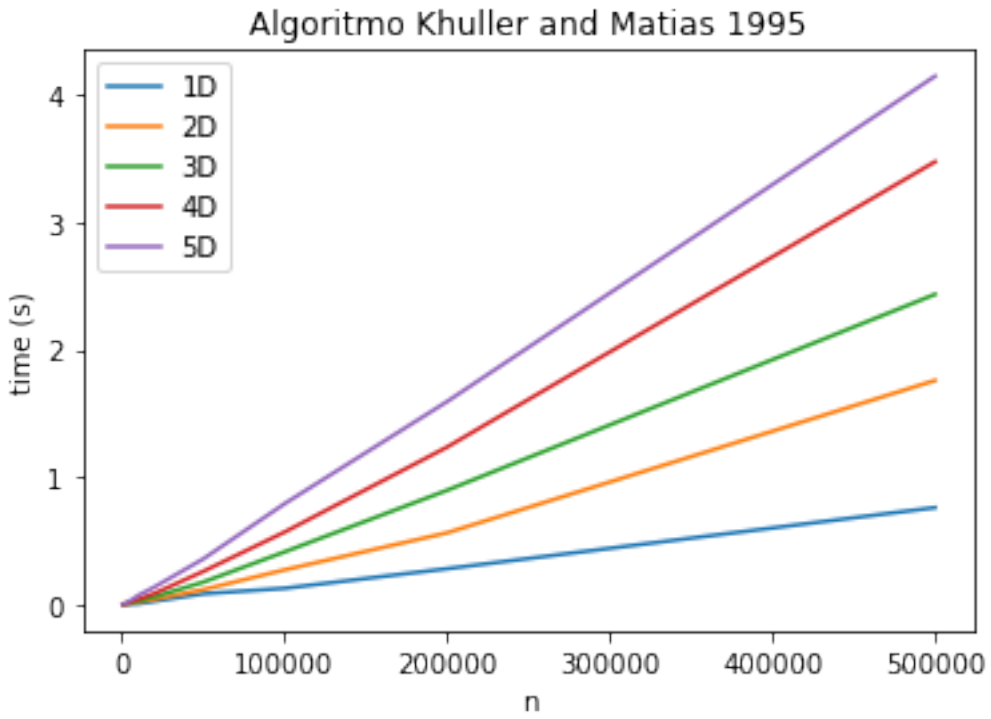
Se comprueba con los valores númericos y con las gráficas su rendimiento $O(n^2)$. Nótese que el rendimiento con las dimensiones es prácticamente insignificante, porque no se calcula la distancia total si la distancia ya supera sin computarse todas las dimensiones entonces se aborta el cálcula completo de la distancia. Esto hace que en la mayoría de los casos, se aborte el cálculo de la distancia con la primera dimensión operada.

[ ]:
```
import timeit
import matplotlib.pyplot as plt
x = [500, 1000, 2000, 5000, 10000, 20000, 50000, 100000, 200000, 500000]
y = {}
for dim in [1,2,3,4,5]:
    print(f"Dimensiones: {dim}D")
    ydim = []
    for num in x:
        sets = 10 if num < 100 else 5
        reps = 1000*1000//(sets*num*num)+1
        t_avg = 0
        for r in range(sets):
            puntos=puntos_aleatorios(num,dim,10000000)
            t = timeit.timeit(lambda:puntos_cercanos(puntos), number=reps)
            t_avg += t
        t_avg /= sets*reps
        ydim.append(t_avg)
        k = t_avg / num
```

```
        print(f"  puntos: {num} tiempo: {t_avg} s ({k}*n s)")
    y[dim] = ydim
plt.plot(x,y[1],x,y[2],x,y[3],x,y[4],x,y[5])
plt.xlabel("n")
plt.ylabel("time (s)")
plt.title('Algoritmo Khuller and Matias 1995')
plt.legend(["1D", "2D", "3D", "4D","5D"])
plt.show()
```

```
Dimensiones: 1D
  puntos: 500 tiempo: 0.0007194731908384711 s (1.4389463816769423e-06*n s)
  puntos: 1000 tiempo: 0.0008883596048690379 s (8.88359604869038e-07*n s)
  puntos: 2000 tiempo: 0.0026449772005435078 s (1.3224886002717538e-06*n s)
  puntos: 5000 tiempo: 0.004767443588934839 s (9.534887177869678e-07*n s)
  puntos: 10000 tiempo: 0.00684828560333699 s (6.848285603336989e-07*n s)
  puntos: 20000 tiempo: 0.025953214202309026 s (1.2976607101154513e-06*n s)
  puntos: 50000 tiempo: 0.08472580059315078 s (1.6945160118630156e-06*n s)
  puntos: 100000 tiempo: 0.12773439679876902 s (1.2773439679876901e-06*n s)
  puntos: 200000 tiempo: 0.2817745179927442 s (1.408872589963721e-06*n s)
  puntos: 500000 tiempo: 0.7610623129992746 s (1.5221246259985492e-06*n s)
Dimensiones: 2D
  puntos: 500 tiempo: 0.001209714391734451 s (2.419428783468902e-06*n s)
  puntos: 1000 tiempo: 0.00180421780096367 s (1.80421780096367e-06*n s)
  puntos: 2000 tiempo: 0.003513191995443776 s (1.756595997721888e-06*n s)
  puntos: 5000 tiempo: 0.00882495759287849 s (1.7649915185756982e-06*n s)
  puntos: 10000 tiempo: 0.020355246803956107 s (2.0355246803956105e-06*n s)
  puntos: 20000 tiempo: 0.04092427259893157 s (2.046213629946578e-06*n s)
  puntos: 50000 tiempo: 0.11754602300934494 s (2.350920460186899e-06*n s)
  puntos: 100000 tiempo: 0.27278909520246086 s (2.727890952024609e-06*n s)
  puntos: 200000 tiempo: 0.5637808156083338 s (2.818904078041669e-06*n s)
  puntos: 500000 tiempo: 1.7590632137958893 s (3.5181264275917784e-06*n s)
Dimensiones: 3D
  puntos: 500 tiempo: 0.0013434371910989284 s (2.686874382197857e-06*n s)
  puntos: 1000 tiempo: 0.002263108611805364 4 s (2.2631086118053644e-06*n s)
  puntos: 2000 tiempo: 0.004549127200152725 s (2.2745636000763624e-06*n s)
  puntos: 5000 tiempo: 0.012447042198618873 s (2.4894084397237745e-06*n s)
  puntos: 10000 tiempo: 0.026651646796381 1 s (2.66516467963811e-06*n s)
  puntos: 20000 tiempo: 0.06046541080577299 s (3.0232705402886496e-06*n s)
  puntos: 50000 tiempo: 0.17647363059804774 s (3.5294726119609548e-06*n s)
  puntos: 100000 tiempo: 0.4136670149920974 s (4.1366701499209744e-06*n s)
  puntos: 200000 tiempo: 0.8973676604044158 s (4.486838302022079e-06*n s)
  puntos: 500000 tiempo: 2.434654117608443 s (4.869308235216886e-06*n s)
Dimensiones: 4D
  puntos: 500 tiempo: 0.0019135600072331727 s (3.827120014466345e-06*n s)
  puntos: 1000 tiempo: 0.003099642199231311 7 s (3.099642199231312e-06*n s)
  puntos: 2000 tiempo: 0.006686790997628122 s (3.343395498814061e-06*n s)
  puntos: 5000 tiempo: 0.017640584596665575 s (3.528116919333115e-06*n s)
```

```
  puntos: 10000 tiempo: 0.04098910100292415 s (4.098910100292415e-06*n s)
  puntos: 20000 tiempo: 0.08730122579145246 s (4.365061289572623e-06*n s)
  puntos: 50000 tiempo: 0.2606032107956707 s (5.212064215913415e-06*n s)
  puntos: 100000 tiempo: 0.5690872673934791 s (5.690872673934792e-06*n s)
  puntos: 200000 tiempo: 1.2365418736007996 s (6.182709368003998e-06*n s)
  puntos: 500000 tiempo: 3.475160777597921 s (6.950321555195842e-06*n s)
Dimensiones: 5D
  puntos: 500 tiempo: 0.002713255595881492 s (5.4265111917629845e-06*n s)
  puntos: 1000 tiempo: 0.00594784920103848 s (5.94784920103848e-06*n s)
  puntos: 2000 tiempo: 0.009075042401673273 s (4.537521200836365e-06*n s)
  puntos: 5000 tiempo: 0.028871228406205773 s (5.774245681241154e-06*n s)
  puntos: 10000 tiempo: 0.07199473539949394 s (7.199473539949944e-06*n s)
  puntos: 20000 tiempo: 0.13809878140455112 s (6.904939070227556e-06*n s)
  puntos: 50000 tiempo: 0.35861954819993114 s (7.172390963998623e-06*n s)
  puntos: 100000 tiempo: 0.7917008315969725 s (7.917008315969724e-06*n s)
  puntos: 200000 tiempo: 1.5938204831967595 s (7.969102415983798e-06*n s)
  puntos: 500000 tiempo: 4.146674714004621 s (8.293349428009241e-06*n s)
```



Se comprueba con los valores númericos y con las gráficas su rendimiento $O(n)$. Nótese que cuando llega a un tamaño dado empeora ligeramente peor que de forma lineal (se ve en los resultados numéricos). Este efecto se lo achaco a que el problema se sale del tamaño de la caché del microprocesador, que cada vez resulta menos eficaz por el problema ser más grande. Además, a priori, el algoritmo parece que debería empeorar el rendimiento exponencialmente con el número de dimensiones, teniendo en cuenta que el número de vecinos de Moore es 3^k. No obstante, una estructura

jerárquica inteligente de las hash-table ha permitido acelerarlo mucho, de forma que los vecinos no existentes en una dimensión no se buscan en otra. Esto reduce muchísimo el cálculo y hace crecer la complejidad del algoritmo de forma mucho más lenta con el número de dimensiones.