Instituto Politécnico Nacional

Escuela Superior de Cómputo

Análisis de algoritmos

Profesora. **Luz María Sánchez García**

PRÁCTICA 4

ALGORITMOS DIVIDE Y VENCERÁS

SEGUNDA PARTE

Grupo: 3CM13

Equipo: CODEART

Integrantes:

1. Shim Kyuseop
2. Ortiz Jiménez José Antonio
3. Quintero Maldonado Iván

Fecha de entrega: 11 de abril de 2022

Planteamiento del problema

Calcular pow (x,n)

La función pow(x,n) es para calcular valor de x^n. Una solución simple para calcular pow(x, n) es multiplicar x exactamente n veces. Podemos hacer eso usando un bucle for simple. Sin embargo, también se puede resolver implementando divide y vencerás

Algoritmo de Karatsuba

El algoritmo de Karatsuba es un algoritmo de multiplicación rápida que utiliza un enfoque de divide y vencerás para multiplicar dos números. Fue descubierto por Anatoly Karatsuba en 1962. Si tenemos que multiplicar dos números de n dígitos x y y, esto se puede hacer con las siguientes operaciones, asumiendo que B es la base de m y m < n (por ejemplo: m = n/2). Primero, ambos números, x y y, se pueden representar como x1, x2 y y1, y2 como la siguiente fórmula.

$$x = x1 * B^m + x2$$

$$y = y1 * B^m + y2$$

$$xy = (x1 * B^m + x2)(y1 * B^m + y2)$$

$$=> xy = x1 * y1 * B^(2m) + x1 * y2 * B^m + x2 * y1 * B^m + x2 * y2$$

Hay 4 subproblemas: X1 * Y1, X1 * Y2, X2 * Y1 y X2 * Y2. Dejamos que

$$a = x1 * y1, b = x1 * y2 + x2 * y1 \text{ and } c = x2 * y2$$

Entonces tenemos

$$xy = a * B^(2m) + b * B^m + c$$

Y finalmente

$$b = (x1 + x2)(y1 + y2) - a - c$$

Par de puntos más cercanos

Para definir distancia entre puntos pares, utilizamos formula

$$\|pq\| = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$$

Y con base esto, se puede elaborar algoritmo que define cuando este distancia es más pequeño. Con algoritmo de divide y vencerá, se utiliza siguientes pasos.

1) Encuentre el punto medio en arreglo ordenado, podemos tomar P[n/2] como punto medio.
2) Divide el arreglo dado en dos mitades. El primer subarreglo contiene puntos de P[0] a P[n/2]. El segundo subarreglo contiene puntos de P[n/2+1] a P[n-1].
3) Encuentre recursivamente las distancias más pequeñas en ambos subarreglos. Sean las distancias dl y dr. Encuentre el mínimo de dl y dr. Sea el mínimo d.

Actividades

1. El programa de los algoritmos Divide y vencerás seleccionados

   Consulte las secciones Prueba y Anexo

2. El programa debe utilizar funciones recursivas

   Consulte las secciones Prueba y Anexo

3. **Análisis temporal**

   pow (x,n)

   La ecuación de recurrencia es

```python
def pow(number:int, target:int):
  if target == 0:
    return 1
  elif target%2==0:
          return pow(number, target/2)*pow(number, target/2)
  else:
    return number*pow(number, target//2)*pow(number, target//2)
```

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

   La complejidad temporal en Big O es

$$O(\log n)$$

Algoritmo de Karatsuba

```
def multiply_by_karatsuba(num_x: int, num_y: int):
  if (num_x < 10) or (num_y < 10) :
    return num_x * num_y
  else:
    max_length = max( functions.number_length(num_x),  functions.number_length(num_y)  )
    max_length_avr = max_length / 2
    max_length_avr = math.floor( max_length_avr )

    print( num_x, num_y, max_length_avr )
    a = num_x / 10**(max_length_avr)
    b = num_x % 10**(max_length_avr)
    c = num_y / 10**(max_length_avr)
    d = num_y % 10**(max_length_avr)
    ac = multiply_by_karatsuba(a,c)
    bd = multiply_by_karatsuba(b,d)
    ad_plus_bc = multiply_by_karatsuba(a+b,c+d) - ac - bd
    prod = ac * 10**(2*max_length_avr) + (ad_plus_bc * 10**max_length_avr) + bd
    return prod
```

La ecuación de recurrencia es

$$T(n) = 3 \cdot T\left(\frac{n}{2}\right) + O(n)$$

$$a = 3, \quad b = 2, \quad k = 1$$

$$Como\ a = 3 > 2 = b^k$$

$$T(n) = O\left(n^{\log_b a}\right) = O(n^{\log_2 3}) \cong O(n^{1.585})$$

Par de puntos más cercanos

```
class Point():
        def __init__(self, x, y):
                self.x = x
                self.y = y
P = [Point(2, 3), Point(12, 30),
        Point(40, 50), Point(5, 1),
        Point(12, 10), Point(3, 4)]
n = len(P)


# Change Recursion maximum
sys.setrecursionlimit(10000)


def bruteForce(P, n):
        min_val = float('inf')
        for i in range(n):
                for j in range(i + 1, n):
                        if functions.dist(P[i], P[j]) < min_val:
                                min_val = functions.dist(P[i], P[j])

        return min_val


def stripClosest(strip, size, d):
        min_val = d
        for i in range(size):
                j = i + 1
                while j < size and (strip[j].y -
                                                        strip[i].y) < min_val:
                        min_val = functions.dist(strip[i], strip[j])
                        j += 1

        return min_val
```

```
def closestUtil(P, Q, n):
        if n <= 3:
                return bruteForce(P, n)

        mid = n // 2
        midPoint = P[mid]

        Pl = P[:mid]
        Pr = P[mid:]

        dl = closestUtil(Pl, Q, mid)
        dr = closestUtil(Pr, Q, n - mid)

        d = min(dl, dr)

        stripP = []
        stripQ = []
        lr = Pl + Pr
        for i in range(n):
                if abs(lr[i].x - midPoint.x) < d:
                        stripP.append(lr[i])
                if abs(Q[i].x - midPoint.x) < d:
                        stripQ.append(Q[i])

        stripP.sort(key = lambda point: point.y) #<-- REQUIRED
        min_a = min(d, stripClosest(stripP, len(stripP), d))
        min_b = min(d, stripClosest(stripQ, len(stripQ), d))

        return min(min_a,min_b)


def closest(P, n):
        P.sort(key = lambda point: point.x)
        Q = copy.deepcopy(P)
        Q.sort(key = lambda point: point.y)

        return closestUtil(P, Q, n)
```

Se necesita O(nlogn) para ordenamiento. Luego, necesita O(n) para buscar puntos más cercanos en strip[]. Finalmente se requiere O(nlogn) otra vez cuando combinan la solución.

La ecuación de recurrencia es

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n \cdot logn)$$

La complejidad temporal en Big O es

$$O(n \cdot \log^2 n)$$

4. Calcular la cantidad de instrucciones que tarda carda algoritmo

pow (x,n)

k	Instrucciones
100	2
200	2.301029996
300	2.477121255
400	2.602059991
500	2.698970004
600	2.77815125
700	2.84509804
800	2.903089987
900	2.954242509
1000	3
5000	3.698970004
10000	4
100000	5

Algoritmo de Karatsuba

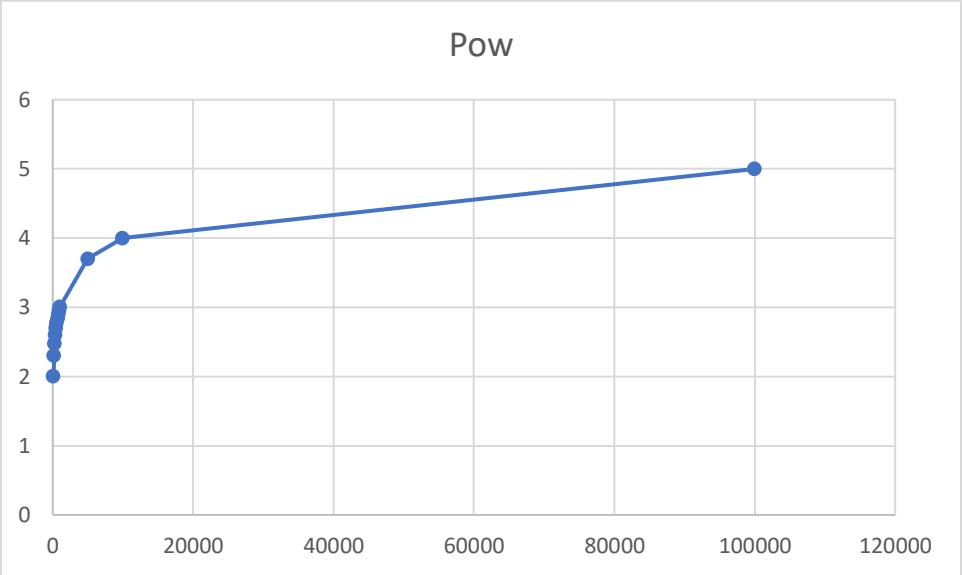
k	Instrucciones
100	1478.852982
200	4436.558946
300	8436.150103
400	13309.67684
500	18956.85322
600	25308.45031
700	32312.73391
800	39929.03052
900	48124.20803
1000	56870.55966
5000	729002.0476
10000	2187006.143
100000	84103196.75

Par de puntos más cercanos

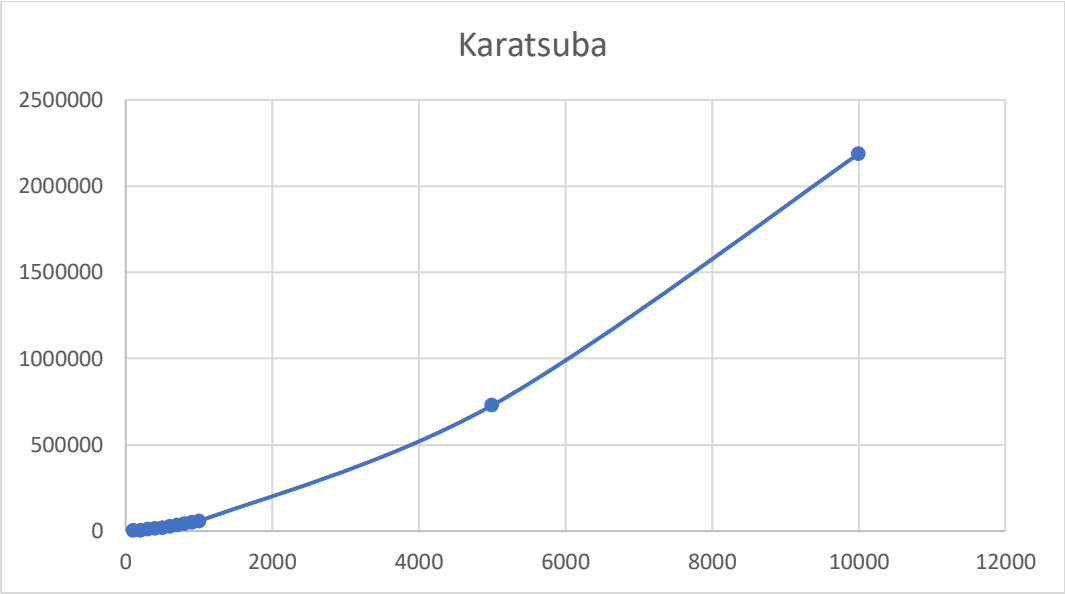
K	INSTRUCCIONES
100	4414.082507
200	11685.70749
300	20314.03711
400	29886.49993
500	40192.64392
600	51102.65665
700	62527.77404
800	74403.16977
900	86679.27114
1000	99316.85641
5000	754939.3776
10000	1765633.003

5. Gráficas de comportamiento temporal de cada algoritmo

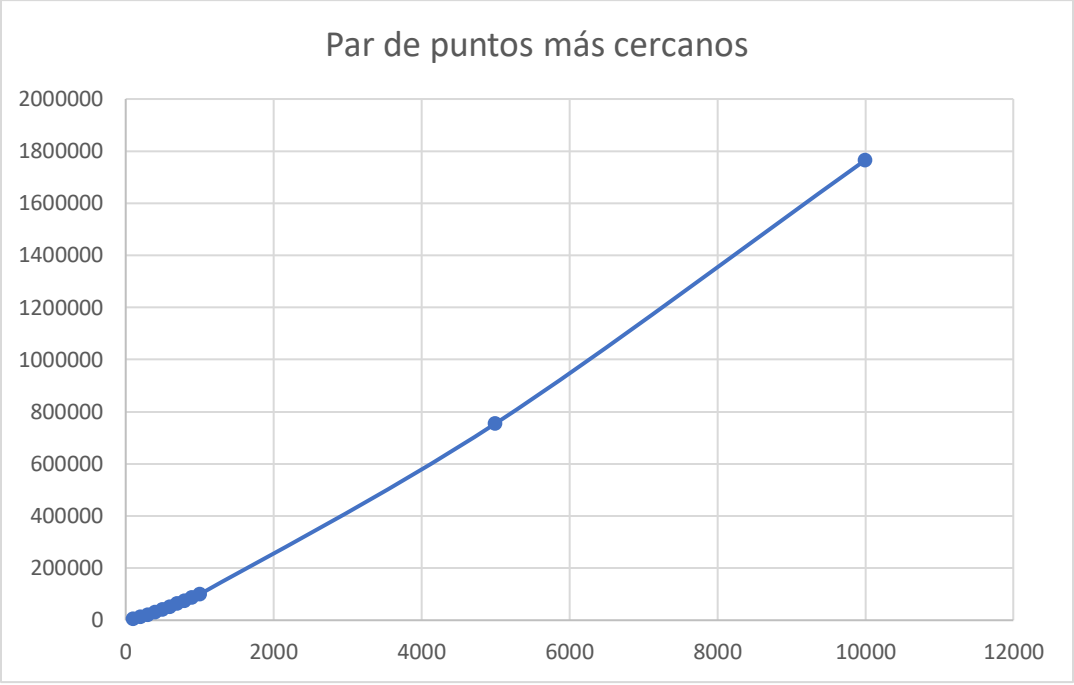
pow (x,n)



Algoritmo de Karatsuba



Par de puntos más cercanos



6. Análisis espacial

pow (x,n)

Su complejidad espacial es O(1), porque solo se usa una variable.

Algoritmo de Karatsuba

Su complejidad espacial es O(n), porque se usa un arreglo cuyo longitud varia dependiendo entrada n

Par de puntos más cercanos

Su complejidad espacial es O(n).

7. Calcular cantidad de celdas de memorias necesarias para cada algoritmo

pow (x,n)

K	CELDAS
100	1
200	1
300	1
400	1
500	1
600	1
700	1
800	1
900	1
1000	1
5000	1
10000	1

Algoritmo de Karatsuba

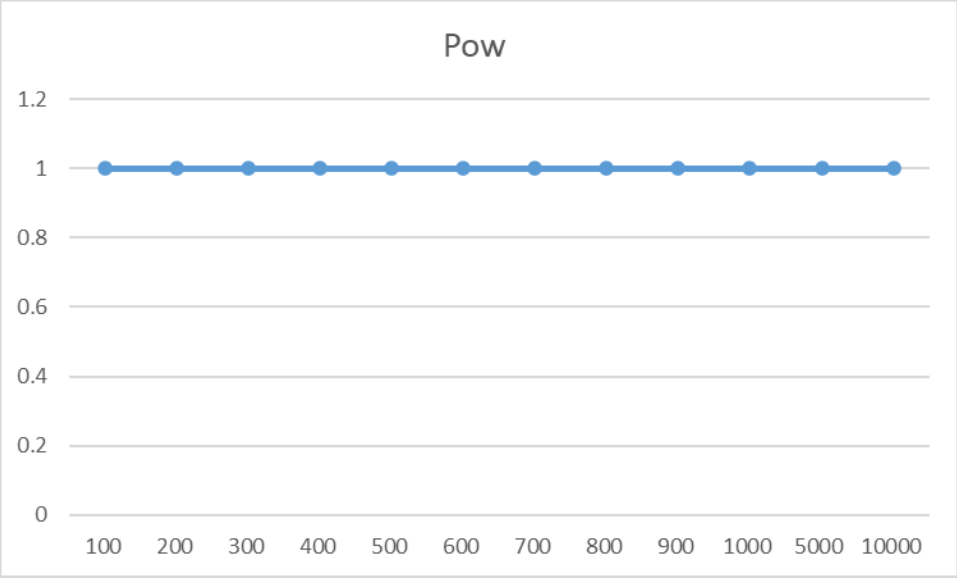
K	CELDAS
100	100
200	200
300	300
400	400
500	500
600	600
700	700
800	800
900	900
1000	1000
5000	5000
10000	10000

Par de puntos más cercanos

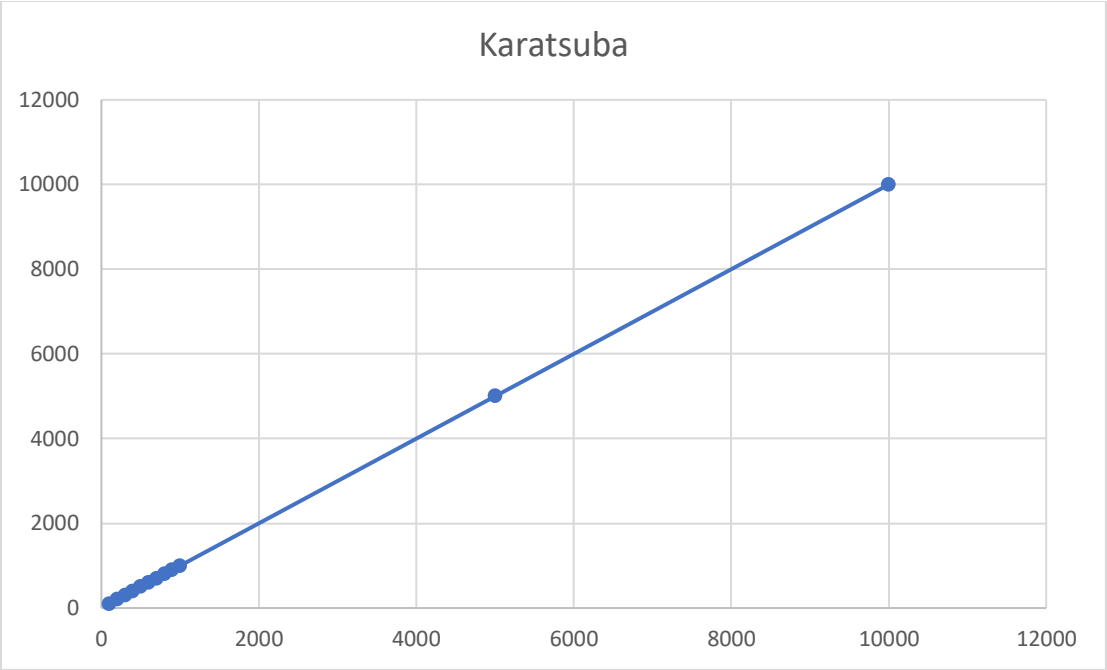
K	CELDAS
100	100
200	200
300	300
400	400
500	500
600	600
700	700
800	800
900	900
1000	1000
5000	5000
10000	10000

8. Gráficas de comportamiento espacial de cada algoritmo

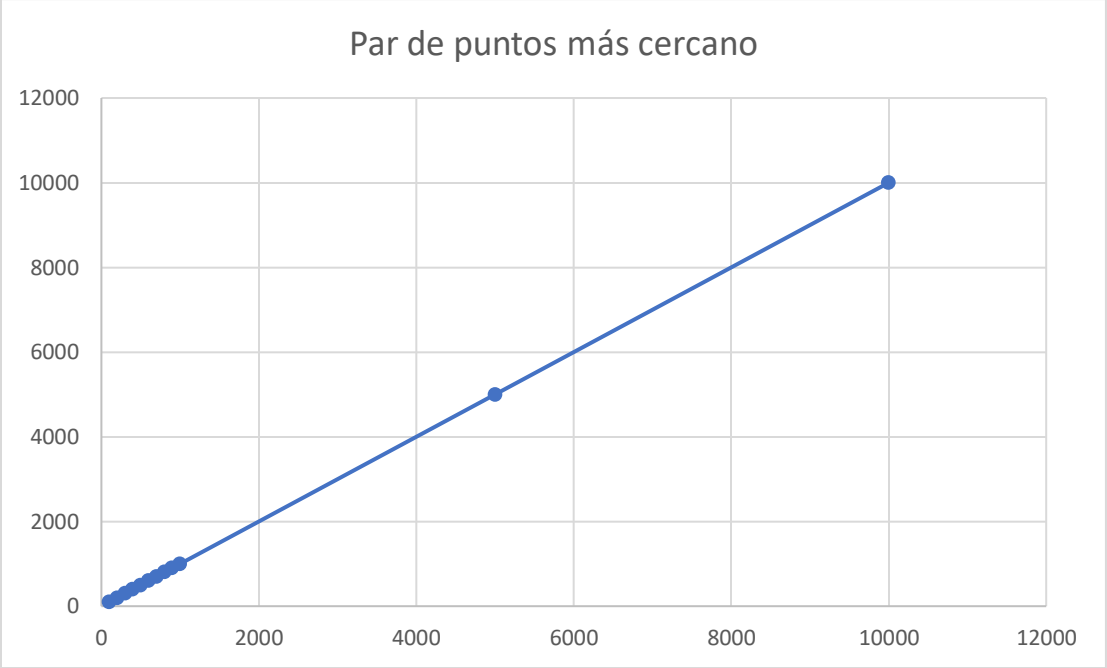
pow (x,n)



Algoritmo de Karatsuba



Par de puntos más cercanos



9. Indique el entorno controlado para realizar las prubeas experimentales

- Sistema operativo basado en Linux
- Python 3

10. Preguntas

a) A parte de la estrategia Divide y vencerás para cada algoritmo ¿Existe o existen otras estrategias por las cuales se resuelve?

Si, la multiplicación ya existen más algoritmos más eficientes como Van der Hoeven

b) ¿Cuál de los tres algoritmos presenta mayor complejidad temporal?

Par de puntos más cercanos en n con valor chico, y karatsuba en valor de n más grande.

c) ¿Cuál de los tres algoritmos presenta menor complejidad temporal?

Calculó de pow(x,n)

d) ¿Cuáles son las ventajas de implementar los tres algoritmos con la estrategia Divide y vencerás? ¿Por qué?

Ya que se divide en problemas más simples, se puede resolver más rapido. Por ejemplo, en karatsuba si usamos algoritmo que usamos en calculo, tarda $O(n^2)$) pero con divide y vencera se hace en $O(n^{1.585})$.

e) ¿Qué otros algoritmos Divide y vencerás recomendarías para realizar esta práctica?

Multiplicación de dos polinomios, porque no es tan dificil de entender.

Anexo

Main.py

```python
from typing import List
import math
import copy
import os
import sys
sys.path.append( os.getcwd() + "/mixins")
import functions


# Variables for pow algorithm
number = 2
target = 10

# multiply
number_x: int = 20000000
```

```
number_y: int = 50000000

# A class to represent a Point in 2D plane
class Point():
    def __init__(self, x, y):
        self.x = x
        self.y = y
P = [Point(2, 3), Point(12, 30),
    Point(40, 50), Point(5, 1),
    Point(12, 10), Point(3, 4)]
n = len(P)


# Change Recursion maximum
sys.setrecursionlimit(10000)


def bruteForce(P, n):
    min_val = float('inf')
    for i in range(n):
        for j in range(i + 1, n):
            if functions.dist(P[i], P[j]) < min_val:
                min_val = functions.dist(P[i], P[j])

    return min_val


def stripClosest(strip, size, d):
    min_val = d
    for i in range(size):
        j = i + 1
        while j < size and (strip[j].y -
                            strip[i].y) < min_val:
            min_val = functions.dist(strip[i], strip[j])
            j += 1

    return min_val


def closestUtil(P, Q, n):
    if n <= 3:
        return bruteForce(P, n)

    mid = n // 2
```

```
    midPoint = P[mid]

    Pl = P[:mid]
    Pr = P[mid:]

    dl = closestUtil(Pl, Q, mid)
    dr = closestUtil(Pr, Q, n - mid)

    d = min(dl, dr)

    stripP = []
    stripQ = []
    lr = Pl + Pr
    for i in range(n):
        if abs(lr[i].x - midPoint.x) < d:
            stripP.append(lr[i])
        if abs(Q[i].x - midPoint.x) < d:
            stripQ.append(Q[i])

    stripP.sort(key = lambda point: point.y) #<-- REQUIRED
    min_a = min(d, stripClosest(stripP, len(stripP), d))
    min_b = min(d, stripClosest(stripQ, len(stripQ), d))

    return min(min_a,min_b)


def closest(P, n):
    P.sort(key = lambda point: point.x)
    Q = copy.deepcopy(P)
    Q.sort(key = lambda point: point.y)

    return closestUtil(P, Q, n)


def karatsuba(x, y):
    if x < 3 or y < 3:
        return x * y

    n = max(len(str(x)), len(str(y))) // 2
    p = 10**n

    a, b = divmod(x, p)
    c, d = divmod(y, p)
```

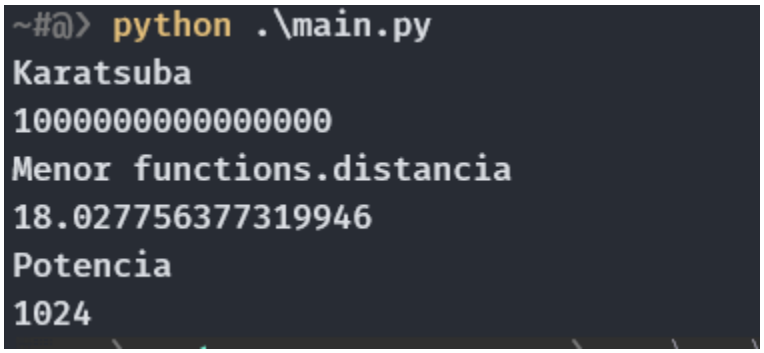
```
    ac = karatsuba(a, c)
    bd = karatsuba(b, d)
    abcd = karatsuba(a+b, c+d) - ac - bd

    return (ac*p + abcd)*p + bd



def pow(number:int, target:int):
  if target == 0:
    return 1
  elif target%2==0:
      return pow(number, target/2)*pow(number, target/2)
  else:
    return number*pow(number, target//2)*pow(number, target//2)




if __name__ == '__main__':
  karatsuba_time = functions.algotithm_time(karatsuba(number_x,
number_y), "Karatsuba" )
  normal_time = functions.algotithm_time( closest(P, n), "Menor
functions.distancia" )

  pow_time = functions.algotithm_time( pow(number, target), "Potencia"
)
```

Functions.py

```
from typing import List
from time import time
import matplotlib.pyplot as plot
import math

def dist(p1, p2):
    return math.sqrt((p1.x - p2.x) *
          (p1.x - p2.x) +
          (p1.y - p2.y) *
          (p1.y - p2.y))


def number_length( num: int ):
  return len(str(num))
```

```python
def algotithm_time( algorithm, title: str ):
  print( title )
  cpu_time_start = time()
  print( algorithm )
  cpu_time_end_karatsuba = (time() - cpu_time_start) * 1000
  return cpu_time_end_karatsuba


# Graph
def graph_bars(label: List[str], data: List[float], title: str,
xlabel: str, ylabel: str):
  plot.bar(label,  data)
  plot.title(title)
  plot.xlabel(xlabel)
  plot.ylabel(ylabel)
  plot.show()
```

Pruebas



Bibliografía

- GeeksforGeeks. (2021, 23 julio). Closest Pair of Points using Divide and Conquer algorithm. Recuperado 11 de abril de 2022, de https://www.geeksforgeeks.org/closest-pair-of-points-using-divide-and-conquer-algorithm/
- GeeksforGeeks. (2022a, enero 12). Activity Selection Problem | Greedy Algo-1. Recuperado 10 de abril de 2022, de https://www.geeksforgeeks.org/activity-selection-problem-greedy-algo-1/

- GeeksforGeeks. (2021b, noviembre 28). Write a program to calculate pow(x,n). Recuperado 11 de abril de 2022, de https://www.geeksforgeeks.org/write-a-c-program-to-calculate-powxn/