Práctica 4: Divide y Vencerás (QuickSort).

Hernández Castellanos César Uriel, Aguilar Garcia Mauricio

Escuela Superior de Cómputo
Instituto Politécnico Nacional, México
*uuriel*12009*u*@*gmail.com*, *mauricio.aguilar.garcia*,90@*gmail.com*

Resumen: En la práctica calculamos analítica y experimentalmente la complejidad el algoritmo de ordenamiento QuickSort, dicho algoritmo implementa el paradigma divide y vencerás, para realizar su análisis se debe analizar también la función Partition. Aparte de estos, realizamos el análisis para el mejor y peor de los casos en ambos algoritmos.

Palabras Clave: Algoritmo, Complejidad, QuickSort, Divide, Vencerás, Partition.

1. Introducción

En la cultura popular, divide y vencerás hace referencia a un refrán que implica resolver un problema difícil, dividiéndolo en partes más simples tantas veces como sea necesario, hasta que la resolución de las partes se torna obvia. La solución del problema principal se construye con las soluciones encontradas.

En las ciencias de la computación, el término divide y vencerás (DYV) hace referencia a uno de los más importantes paradigmas de diseño algorítmico. El método está basado en la resolución recursiva de un problema dividiéndolo en dos o más subproblemas de igual tipo o similar. El proceso continúa hasta que éstos llegan a ser lo suficientemente sencillos como para que se resuelvan directamente. Al final, las soluciones a cada uno de los subproblemas se combinan para dar una solución al problema original.

Esta técnica es la base de los algoritmos eficientes para casi cualquier tipo de problema como, por ejemplo, algoritmos de ordenamiento (quicksort, mergesort, entre muchos otros), multiplicar números grandes (Karatsuba), análisis sintácticos (análisis sintáctico top-down) y la transformada discreta de Fourier.

Por otra parte, analizar y diseñar algoritmos de DyV son tareas que lleva tiempo dominar. Al igual que en la inducción, a veces es necesario sustituir el problema original por uno más complejo para conseguir realizar la recursión, y no hay un método sistemático de generalización.

El nombre divide y vencerás también se aplica a veces a algoritmos que reducen cada problema a un único subproblema, como la búsqueda binaria para encontrar un elemento en una lista ordenada (o su equivalente en computación numérica, el algoritmo de bisección para búsqueda de raíces). Estos algoritmos pueden ser implementados más eficientemente que los algoritmos generales de "divide y vencerás"; en particular, si es usando una serie de

recursiones que lo convierten en simples bucles. Bajo esta amplia definición, sin embargo, cada algoritmo que usa recursión o bucles puede ser tomado como un algoritmo de "divide y vencerás". El nombre decrementa y vencerás ha sido propuesta para la subclase simple de problemas.

La corrección de un algoritmo de "divide y vencerás", está habitualmente probada una inducción matemática, y su coste computacional se determina resolviendo relaciones de recurrencia.[?]

2. Conceptos básicos

2.0.1. QuickSort

El ordenamiento rápido (quicksort en inglés) es un algoritmo creado por el científico británico en computación C. A. R. Hoare, basado en la técnica de divide y vencerás, que permite, en promedio, ordenar n elementos en un tiempo proporcional a n log n.
El algoritmo trabaja de la siguiente forma:

Elegir un elemento de la lista de elementos a ordenar, al que llamaremos pivote.
Resituar los demás elementos de la lista a cada lado del pivote, de manera que a un lado queden todos los menores que él, y al otro los mayores. Los elementos iguales al pivote pueden ser colocados tanto a su derecha como a su izquierda, dependiendo de la implementación deseada. En este momento, el pivote ocupa exactamente el lugar que le corresponderá en la lista ordenada.
La lista queda separada en dos sublistas, una formada por los elementos a la izquierda del pivote, y otra por los elementos a su derecha.
Repetir este proceso de forma recursiva para cada sublista mientras éstas contengan más de un elemento. Una vez terminado este proceso todos los elementos estarán ordenados.
Como se puede suponer, la eficiencia del algoritmo depende de la posición en la que termine el pivote elegido.
En el mejor caso, el pivote termina en el centro de la lista, dividiéndola en dos sublistas de igual tamaño. En este caso, el orden de complejidad del algoritmo es O(n·log n).
En el peor caso, el pivote termina en un extremo de la lista. El orden de complejidad del algoritmo es entonces de $O(n^2)$. El peor caso dependerá de la implementación del algoritmo, aunque habitualmente ocurre en listas que se encuentran ordenadas, o casi ordenadas. Pero principalmente depende del pivote, si por ejemplo el algoritmo implementado toma como pivote siempre el primer elemento del array, y el array que le pasamos está ordenado, siempre va a generar a su izquierda un array vacío, lo que es ineficiente. En el caso promedio, el orden es O(n·log n).
No es extraño, pues, que la mayoría de optimizaciones que se aplican al algoritmo se centren en la elección del pivote.

3. Experimentación y Resultados

Problema uno

i) Mediante gráficas, muestre que el algoritmo Partition tiene complejidad lineal..
ii) Demuestre analíticamente que el algoritmo Partition tiene complejidad lineal.

iii) Mediante gráficas, muestre que el algoritmo QuickSort tiene complejidad (nlogn) (Para obtener sus conclusiones, considere diferentes valores para un arreglo de tamaño n).

iv) Demuestre analíticamente que el algoritmo QuickSort tiene complejidad (nlogn) cuando el pivote divide el arreglo por la mitad.

v) Mediante gráficas, proponga el orden de complejidad de QuickSort cuando todos los elementos del arreglo son distintos y están ordenados en forma decreciente.

```
//Función que declara un arreglo bidimensional y lo ordena.
def main():
    arreglo = [34,32,2,4,34,3,4,3,6,4,5]
    quickSort(arreglo,0,len(arreglo)-1)
    print(arreglo)

//Función que realiza la llamada a main.
if __name__ == '__main__':
    main()


//Función encargada de implementar quicksort
def quickSort(arreglo, p, r):
    if p < r:
        q = partition(arreglo, p, r)
        quickSort(arreglo,p, q-1)
        quickSort(arreglo,q+1,r)


//Función encargada de dividir un arreglo de tamaño n = p-r+1 en dos arreglos de tamaño q-p+1 y r-p
def partition(arreglo, p, r):
    x = arreglo[r]
    j = p - 1
    for i in range(p,r):
        if arreglo[i] < x:
            j += 1
            arreglo[j],arreglo[i] = arreglo[i],arreglo[j]
    j += 1
    arreglo[j],arreglo[r] = arreglo[r],arreglo[j]
    return j
```

Figura 1: Implementación de Quicksort en el lenguaje de programación Python

Inciso uno

En la figura dos se muestra la complejidad del algoritmo de partition, la cual se obtuvo de manera experimental.

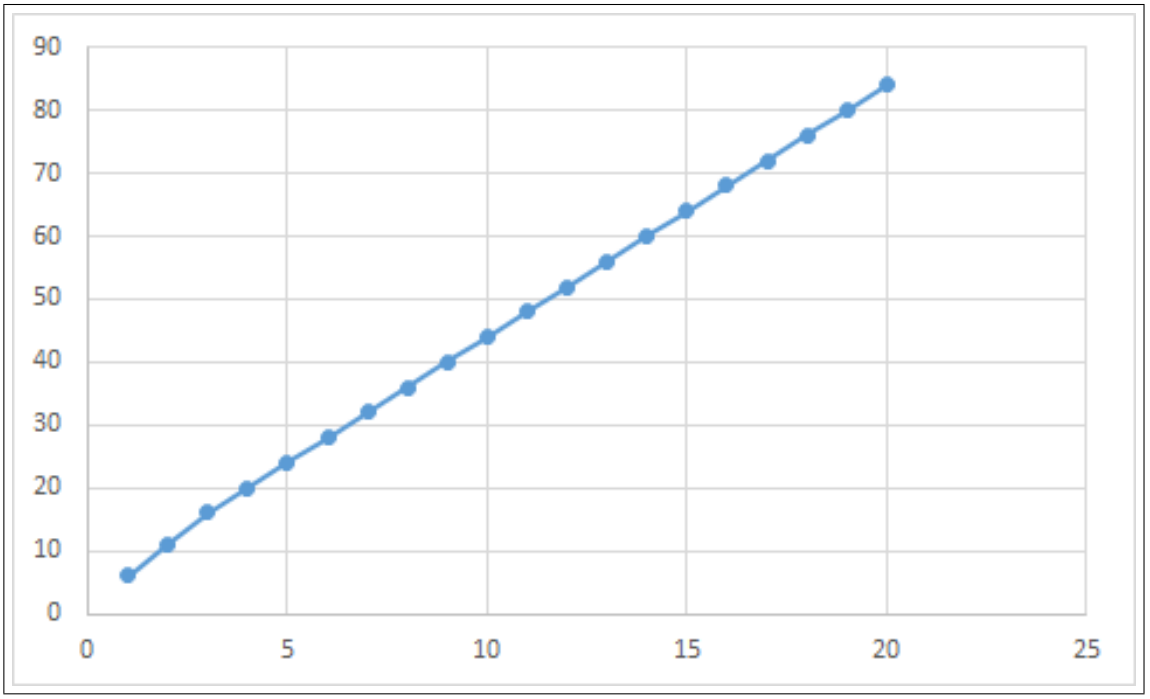

Figura 2: Gráfica de Partition

Inciso dos

$$
\begin{array}{lll}
Partition(A[p,...,r],p,r) & & \\
\quad x = A[r] & \Theta(1) & \\
\quad i = p-1 & & \\
\quad for(j = p; j \leq r-1; j++) & \Theta(r-p+1) = \Theta(n) & \\
\quad\quad if\ A[j] \leq x & & \Theta(n) \\
\quad\quad\quad i++ & & \\
\quad\quad\quad exchange(A[i], A[j]) & & \\
\quad exchange(A[i+1], A[r]) & \Theta(1) & \\
\quad return\ i+1 & &
\end{array}
$$

Figura 3: Demostración formal de la complejidad de partition

En la figura 3 se presenta la demostración formal de la complejidad de partition por medio de teoremas vistos en clase.

Inciso tres

Se tiene implementado el algoritmo de QuickSort:

```python
//Función encargada de implementar quicksort
def quickSort(arreglo, p, r):
    if p < r:
        q = partition(arreglo, p, r)
        quickSort(arreglo,p, q-1)
        quickSort(arreglo,q+1,r)
```

Figura 4: Código de QuickSort

Como salida del código tenemos a la n del tamaño de la entrada y la t la cual es el contador del algoritmo.
Si graficamos estos valores obtenemos la siguiente gráfica la cual nos demuestra que tiene

2	2
4	6
8	24
16	52
32	158
64	435
128	1009
256	2467
512	5217
1024	10861

Figura 5: Salida del programa QuickSort

complejidad $\theta(nlogn)$.
FigureGráfica de la salida del programa.

Inciso cuatro

Código utilizado:

```
void quicksort(int *A,int p,int r){
    if(p<r){                                    \\ C1 = 1
            int q = partition(A,p,r);           \\P(n)
            quicksort(A,p,q-1);                         \\T(?)
            quicksort(A,q,r);                   \\T(?)
    }
}
```

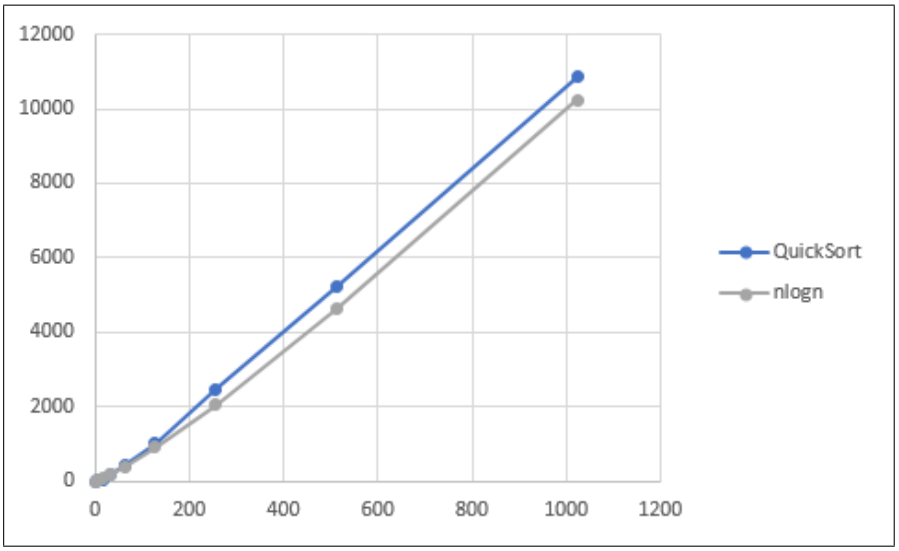

Figura 6: Gráfica de la complejidad de quicksort

$$T(n) = \begin{cases} 1; & p = r, es\ decir, n = 1 \\ 2T\left(\frac{n}{2}\right) + 4n + 8; & p < r, es\ decir, n > 1 \end{cases}$$

$k = \log_2 n$

$$\boldsymbol{T(n)} = \boldsymbol{T(2^k)}$$

$$2[T(2^{k-1})] + 4(2^k) + 8$$

$$2[2[T(2^{k-2})] + 4(2^{k-1}) + 8] + 4(2^k) + 8$$

$$2^2[T(2^{k-2})] + 2[4(2^k)] + 3[8]$$

$$2^2[2[T(2^{k-3})] + 4(2^{k-2}) + 8] + 2[4(2^k)] + 3[8]$$

$$2^3[T(2^{k-3})] + 3[4(2^k)] + 7[8]$$

$$2^3[2[T(2^{k-4})] + 4(2^{k-3}) + 8] + 3[4(2^k)] + 7[8]$$

$$2^4[T(2^{k-4})] + 4[4(2^k)] + 15[8]$$

$$2^{i+1}[T(2^{k-(i+1)})] + (i+1)[4(2^k)] + 8(2^{i+1} - 1)$$

$$2^k[T(2^0)] + (k)[4(2^k)] + 8(2^k - 1)$$

$$n[1] + 4(n\log_2 n) + 8(n-1) \qquad i = k - 1$$

$$4(n\log_2 n) + 9n - 8 \qquad \text{“}k = \log_2 n\text{”}$$

$$n[4(\log_2 n) + 9] - 8$$

$$\boldsymbol{T(n) \in \theta(n\log_2 n)}$$

Inciso cinco

Como podemos ver en la siguiente gráfica la complejidad es $\theta(n^2)$, en la gráfica se ve que se sigue la misma forma de la función aunque se encuentre debajo de la original, se pude notar que la función es justo $(n^2)/2$ pero como sabemos el que se esté dividiendo entre dos puede ser despreciable.

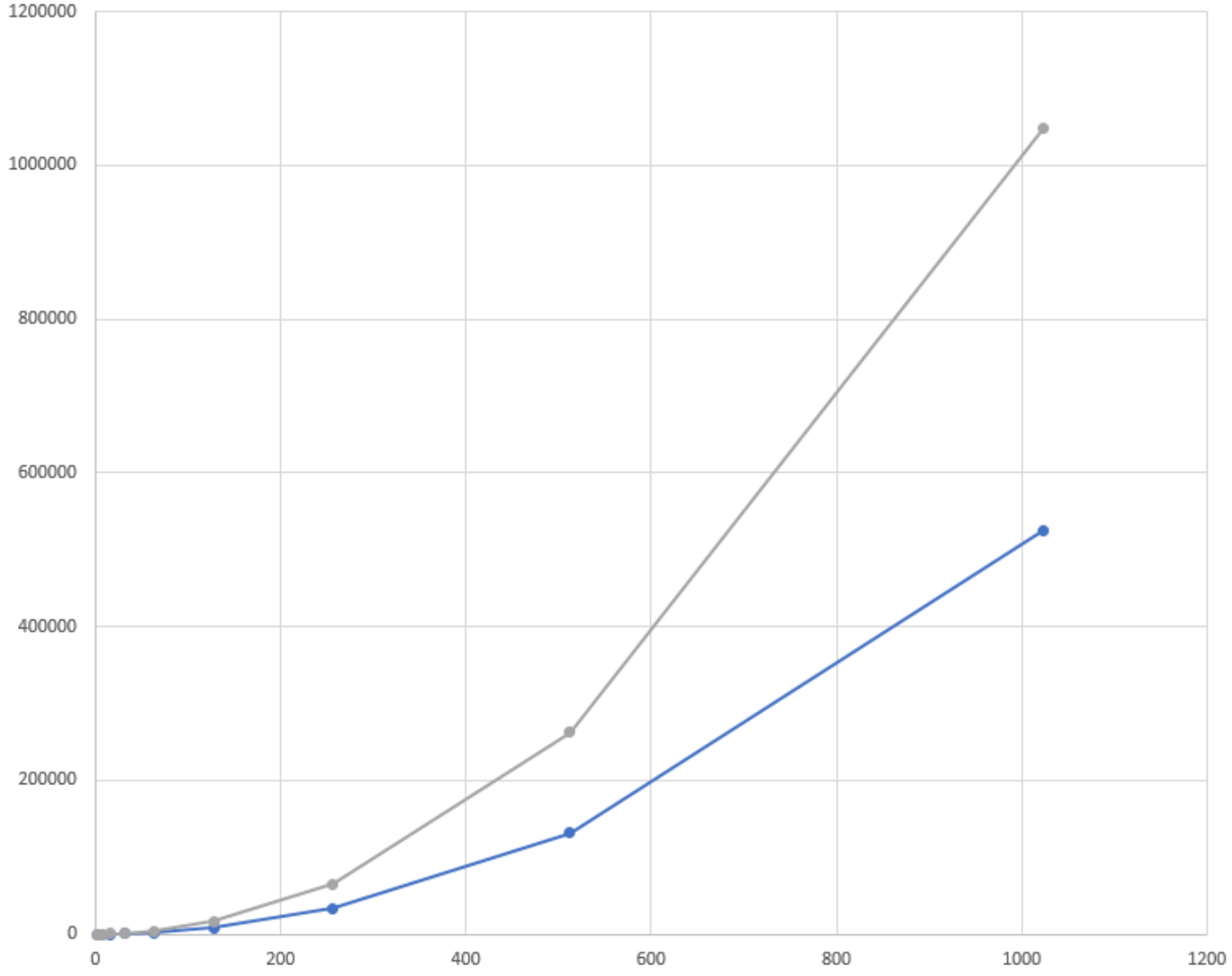

Figura 7: Gráfica de la complejidad de quicksort

En la figura 3 se presenta la demostración formal de la complejidad de partition por medio de teoremas vistos en clase.

4. Conclusiones

Conclusión general

En esta práctica pudimos verificar que hay algoritmos que pueden o no tener la misma complejidad dependiendo de diversos factores, como puede ser el orden de un arreglo, el tamaño de un arreglo, si un número es muy grande, etc. Por ejemplo, el QuickSort es un algoritmo en el que el orden afecta al algoritmo, por lo que puede llegar a ser $O(n^2)$ cuando el arreglo esta ordenado de forma inversa.

Aguilar Garcia Mauricio

En la práctica pudimos notar como el algoritmo QuickSort se divide en el algoritmo partition, el cual tiene complejidad lineal pero al agregarlo al QuickSort y con las diferentes llamadas a su misma función la complejidad se vuelve n log n en su caso promedio, mediante gráficas pudimos notar que el peor caso que es cuando el arreglo tiene números diferentes y ordenados de forma decreciente, el cual se vuelve la complejidad n^2.

Hernández Castellanos César Uriel

Analizando quicksort en el mejor y peor caso se puede verque es uno de los mejores algoritmos de ordenación.
Este algoritmo de ordenación es un ejemplo claro de que elmétodo divide y vencerás es efectivo cuando tienes cantidadesgrandes de datos por trabajar

5. Anexo

5.1. Preguntas

1.¿Qué valor de q retorna Partition cuando todos los elementos en el arreglo A[p, ..., r] tienen el mismo valor?.
El valor que retorna Partition cuando todos los elementos son iguales es igual al p que entra, como se puede observar en la siguiente salida.

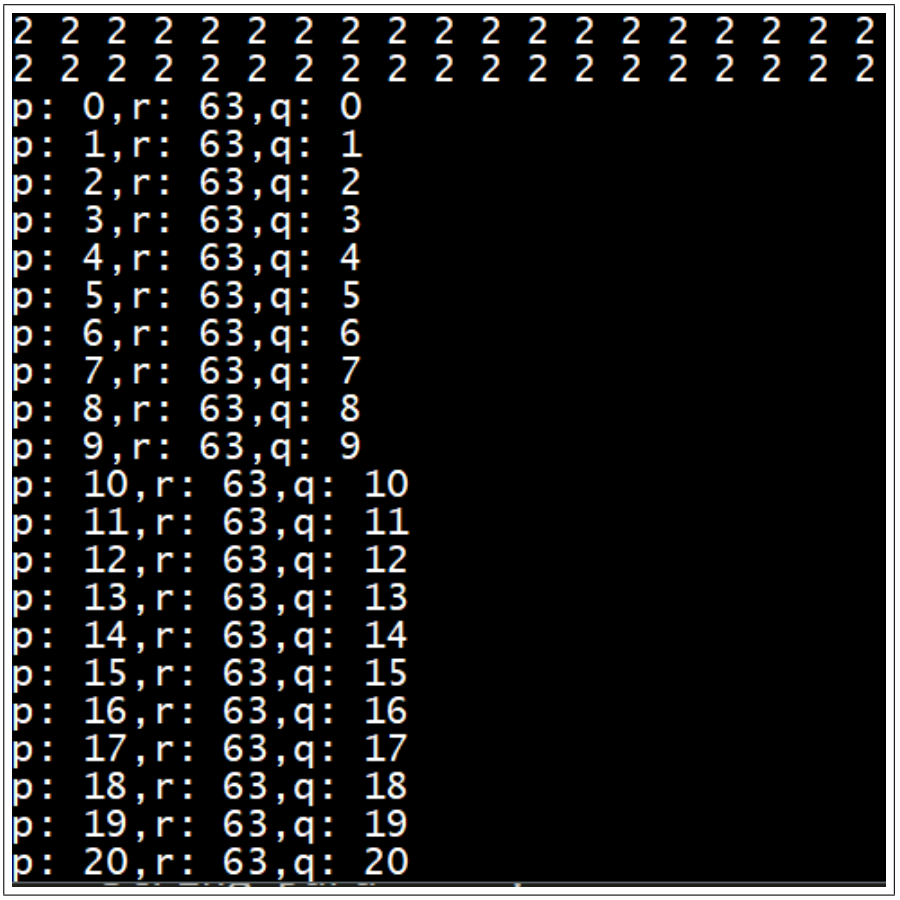

Figura 8: Salida de partition cuando todos los elementos de A son iguales.

2. ¿Cuál es el tiempo de ejecución de QuickSort cuando todos los elementos del arreglo tienen el mismo valor?

Como se puede ver en la experimentación cuando todos los elementos son iguales se llega a que QuickSort es lineal.

6. Bibliografía

[1] 'Algoritmo divide y vencerás' Disponible en https://es.wikipedia.org/wiki/Algoritmo_divide_y_v consultado 12 de Septiembre del 2017

```
1   2,1
2   4,3
3   8,7
4   16,15
5   32,31
6   64,63
7   128,127
8   256,255
9   512,511
10  1024,1023
```

Figura 9: Salida de QuickSort cuando todos los elementos de A son iguales.

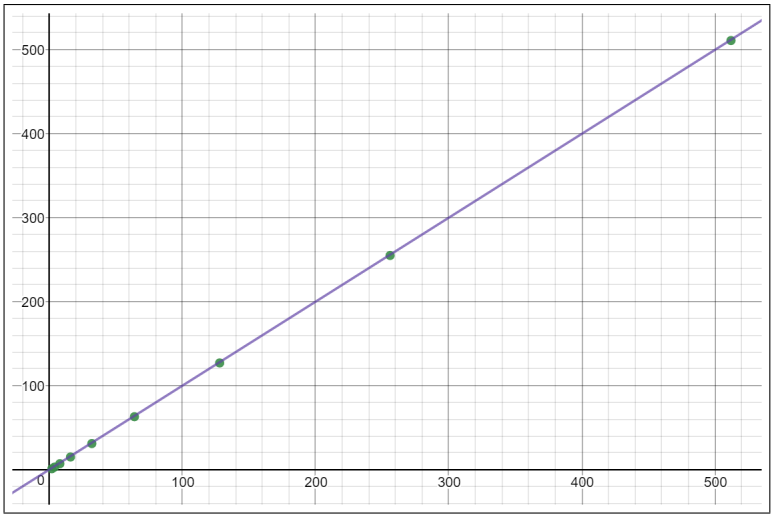

Figura 10: Gráfica de QuickSort cuando todos los elementos son iguales vs gráfica de n.