



UNSA

UNIVERSIDAD NACIONAL DE SAN AGUSTÍN DE AREQUIPA

ANÁLISIS Y DISEÑO DE ALGORITMOS

IMFORME FINAL

NOMBRES:

HUANCA PARQUI ELIZABETH YASMIN

20190748

PROFESOR:

CARLOS EDUARDO ATENCIO TORRES

2021

ÍNDICE

1. INTRODUCCION	2
2. DESARROLLO	2
INSERT VS QUICKSORT VS MERGESORT	2
.....	2
1. En caso de arreglos semiordenados, ¿quién ganará? Haga pruebas para Insert vs Quick vs Mergesort	3
2. Compare Quicksort vs Mergesort. Ejecute una sola prueba con arreglos crecientes y haga una prueba más exhaustiva con arreglos aleatorios. ¿Por qué uno siempre gana en un cierto tipo de prueba que otro?	4
CASO PROMEDIO	8
FIBONACCI	8
1. Discuta sobre qué pasaría si intentáramos encontrar la respuesta en C++ y de forma recursiva? Qué pasaría si lo hiciéramos de forma iterativa?	8
2. ¿Qué pasaría si tratásemos con doubles?	11
3. ¿Qué ocurriría con la memoria si lo intentáramos en python?	12
4. ¿Cómo sería útil si utilizáramos la teoría de Aritmética Modular?	13
5. ¿Cómo sería si utilizáramos la forma matricial de Fibonacci junto a la forma de elevar un número a potencia de n pero utilizando una forma divide y vencerás? - Si, a todo esto, utilice la teoría de aritmética modular.	13
CUADRADOS	13
3. CONCLUSIÓN	14
INSERT VS QUICKSORT VS MERGESORT	14
CASO PROMEDIO	14
FIBONACCI	14
CUADRADOS	14
4. REFERENCIAS	14
5. LINKS DE AVANCE	14

1. INTRODUCCION

En computación y matemáticas un algoritmo de ordenamiento es un algoritmo que pone elementos de una lista o un vector en una secuencia dada por una relación de orden, es decir, el resultado de salida ha de ser una permutación o reordenamiento de la entrada que satisfaga la relación de orden dada. Las relaciones de orden más usadas son el orden numérico y el orden lexicográfico. Ordenamientos eficientes son importantes para optimizar el uso de otros algoritmos (como los de búsqueda y fusión) que requieren listas ordenadas para una ejecución rápida. También es útil para poner datos en forma canónica y para generar resultados legibles por humanos.

2. DESARROLLO

INSERT VS QUICKSORT VS MERGESORT

	INSERT	QUICK	MERGESORT
Conceptos	Al igual que el método de la burbuja va comparando los elementos pero se diferencia que no es necesario recorrer el vector completamente si el elemento es menor se compara inmediatamente con su antecesor.	Consiste simplemente en dividir el vector tomando un pivote como un índice con lo cual se van tomando los elementos comparando con respecto al pivote los cuales los menores lo va dejando a su izquierda así ordenando el vector.	Este método consiste en dividir el vector en varias partes las cuales los compara y coloca los menores a la izquierda y luego todas esas partes los mezcla lo cual compara los elementos colocando siempre el menor a la izquierda
Ventajas	Este algoritmo exhibe un buen rendimiento cuando se trabaja con una lista pequeña de datos.	Este algoritmo es capaz de tratar con una enorme lista de elementos.	Este algoritmo es efectivo para conjuntos de datos que se puedan acceder secuencialmente como arreglos, vectores y listas.
Desventajas	No funciona bien con una lista grande.	Consume muchos recursos para su ejecución.	Requiere un espacio adicional de memoria.

I. En caso de arreglos semiordenados, ¿quién ganará? Haga pruebas para Insert vs Quick vs Mergesort

INSERTSORT

```
Consola de depuración de Microsoft Visual Studio  
SEMIORDENADO  
1 2 3 4 5 6 7 8 9 10  
Execution Time-> 0.004
```

QUICKSORT

```
Consola de depuración de Microsoft Visual Studio  
SEMIORDENADO  
1 2 3 4 5 6 7 8 9 10  
Execution Time-> 0.006
```

MERGESORT

```
Consola de depuración de Microsoft Visual Studio  
SEMIORDENADO  
1 2 3 4 5 6 7 8 9 10  
Execution Time-> 0.006
```

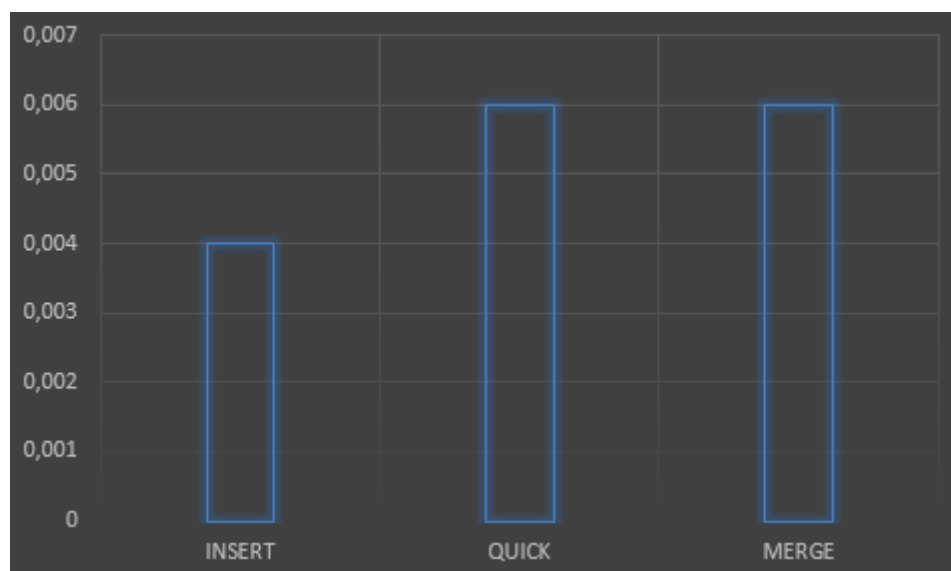


Figura 1.1-Prueba con arreglos semiordenados entre Insert, Quick y Merge

Para un arreglo de 10 elementos de prueba vemos en la figura 1.1 que el algoritmo de ordenamiento por inserción (InsertSort) le toma menos tiempo en ordenar nuestro arreglo, por ende sería el ganador.

2. Compare Quicksort vs Mergesort. Ejecute una sola prueba con arreglos crecientes y haga una prueba más exhaustiva con arreglos aleatorios. ¿Por qué uno siempre gana en un cierto tipo de prueba que otro?

QUICKSORT

```

C:\> Consola de depuración de Microsoft Visual Studio
CRECIENTE
1 2 3 4 5 6 7 8 9 10
Execution Time-> 0.008
    
```

MERGESORT

```

C:\> Consola de depuración de Microsoft Visual Studio
CRECIENTE
1 2 3 4 5 6 7 8 9 10
Execution Time-> 0.005
    
```

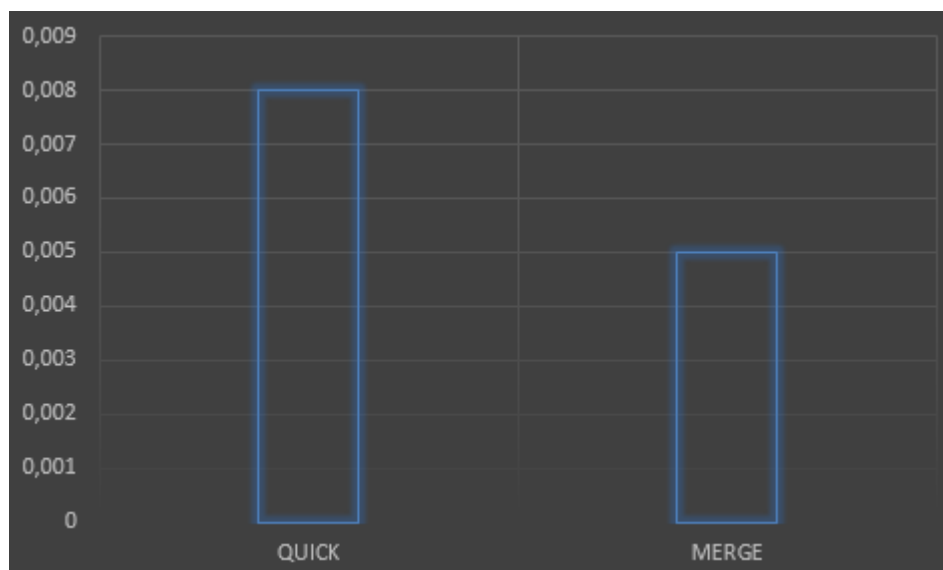


Figura 2.1-Prueba con arreglos crecientes entre Quick y Merge

En esta prueba de 10 Arreglos ingresados de forma creciente vemos en la figura 2.1 que Merge Sort gana a QuickSort.

Elementos	QUICK	MERGE
10	0,001	0,005
20	0,006	0,005
30	0,006	0,007
40	0,013	0,005
50	0,01	0,017
60	0,016	0,006
70	0,012	0,01
80	0,018	0,01
90	0,027	0,014
100	0,035	0,022
500	0,164	0,103
1000	0,21	0,16

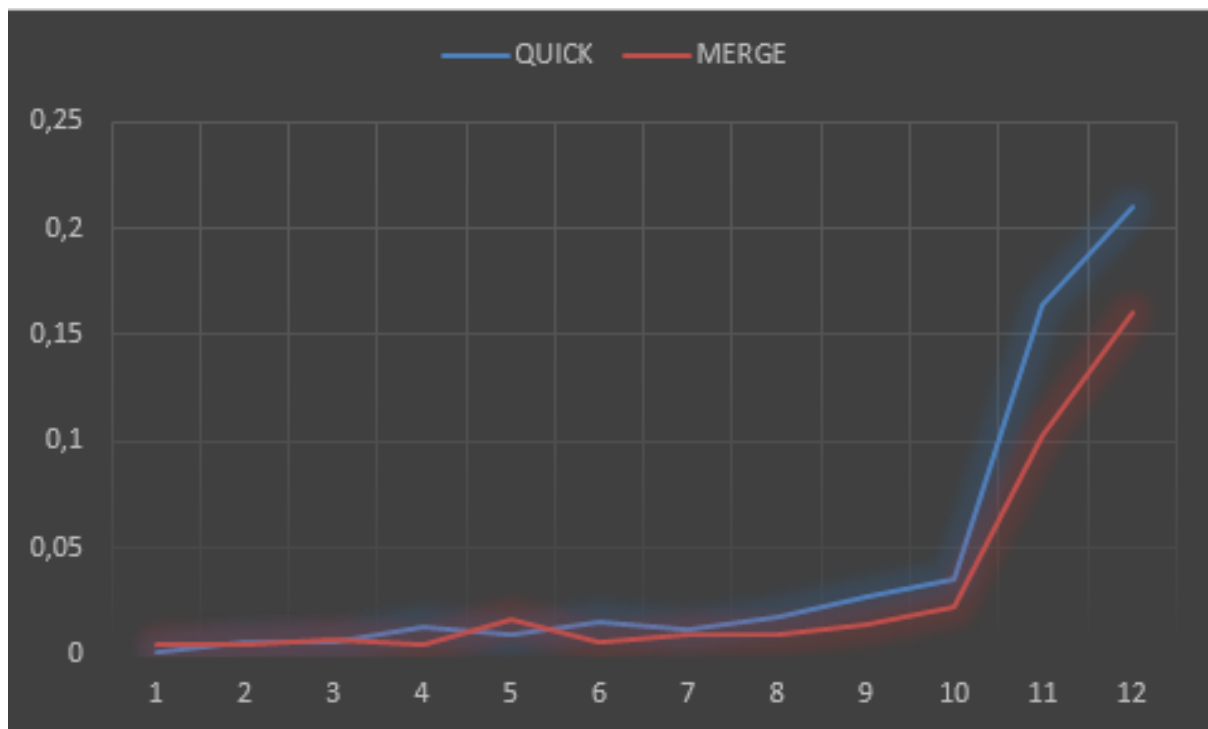


Figura 2.2-Prueba exhaustiva entre Quick y Merge

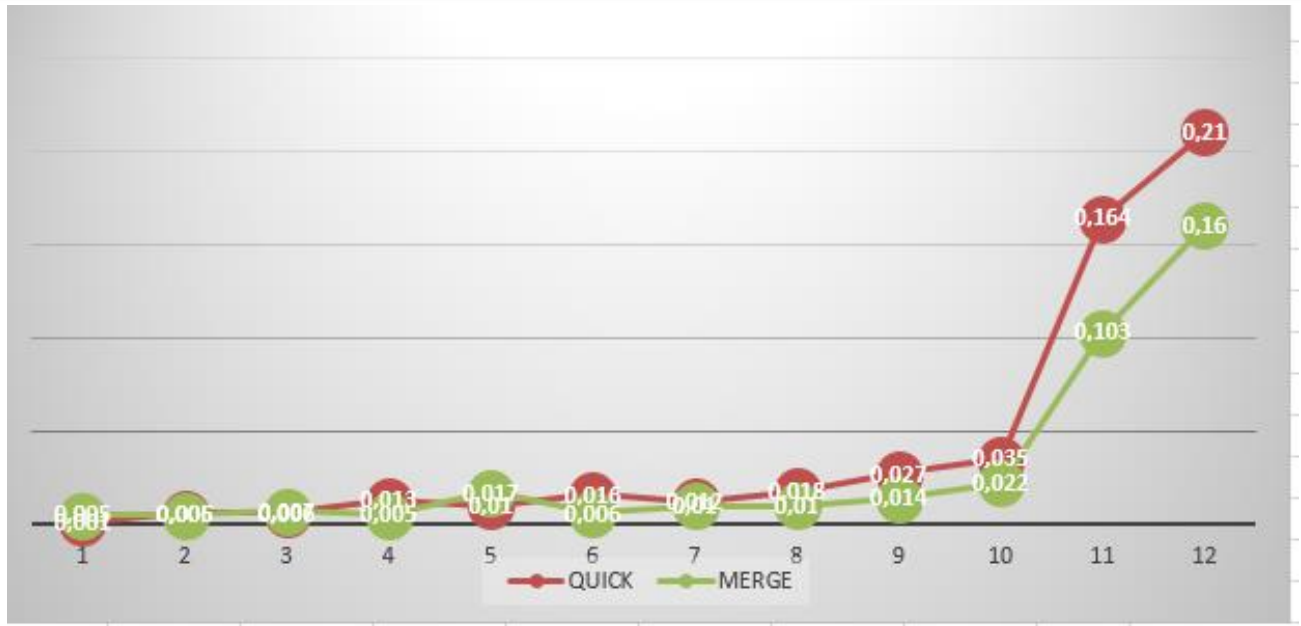


Figura 2.3 Prueba quick vs merge

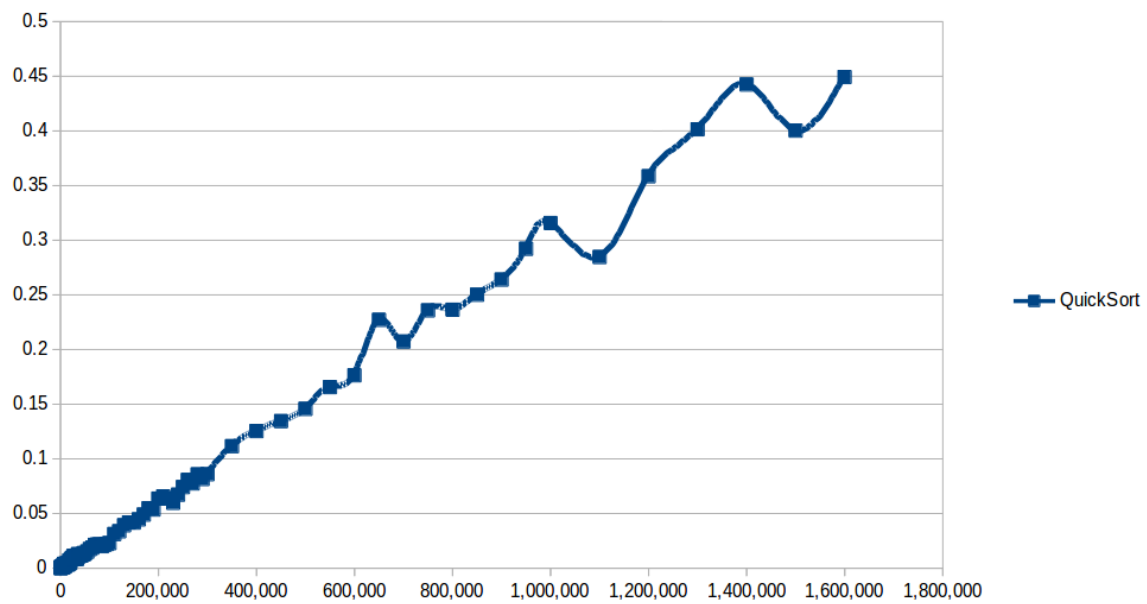


Figura 2.4 – Quick sort

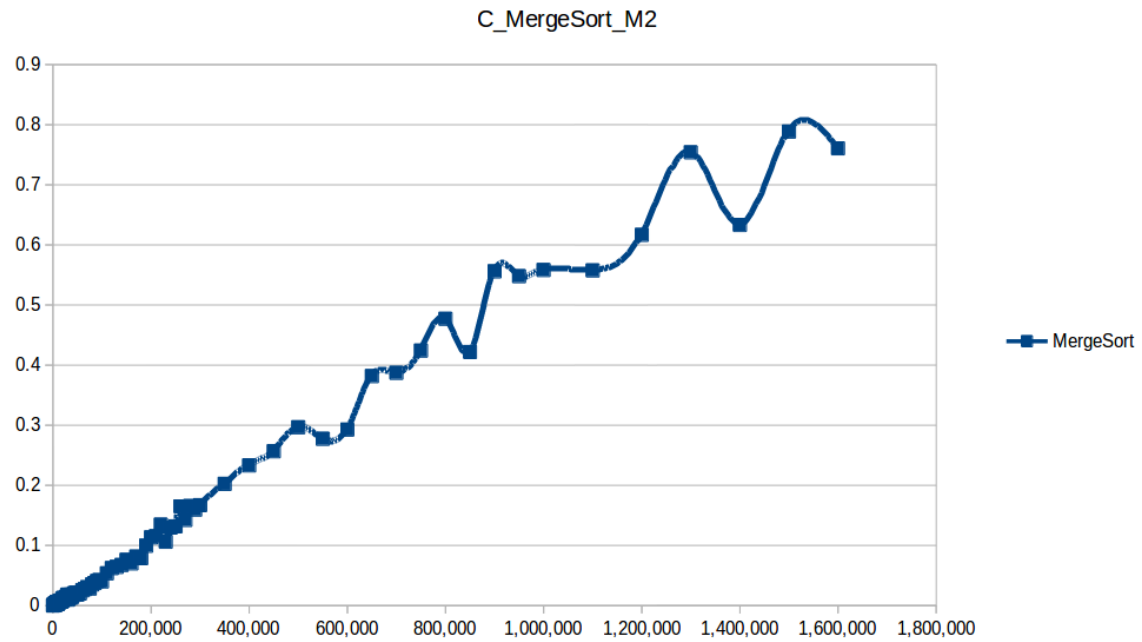


Figura 2.5 – Merge sort

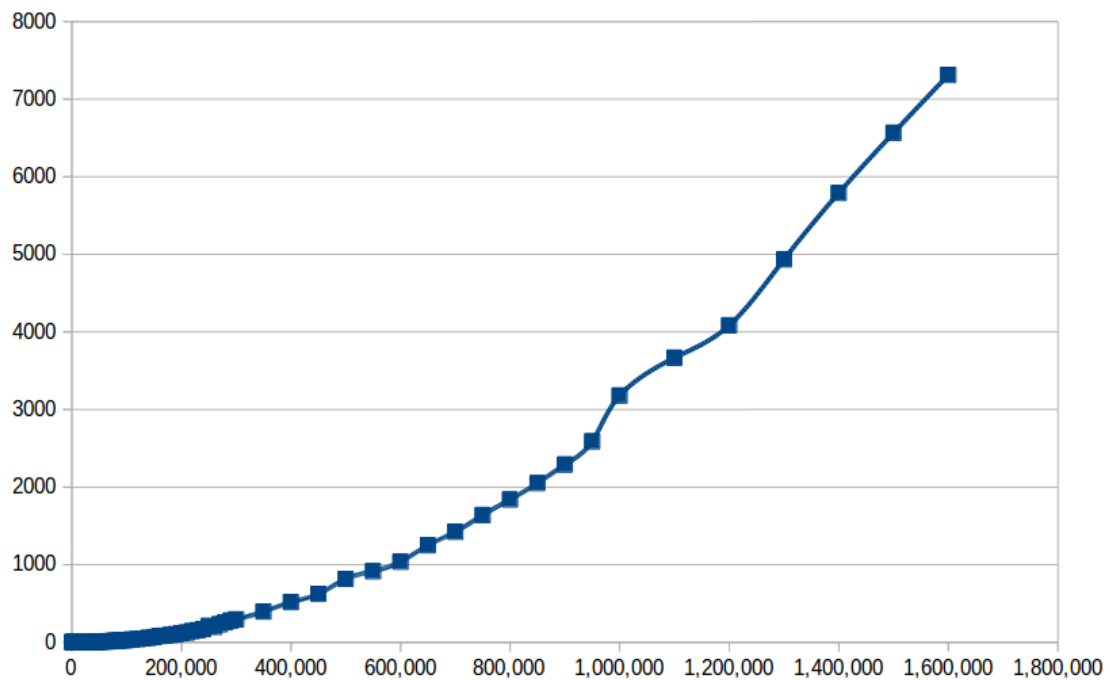


Figura 2.6 – Insert sort

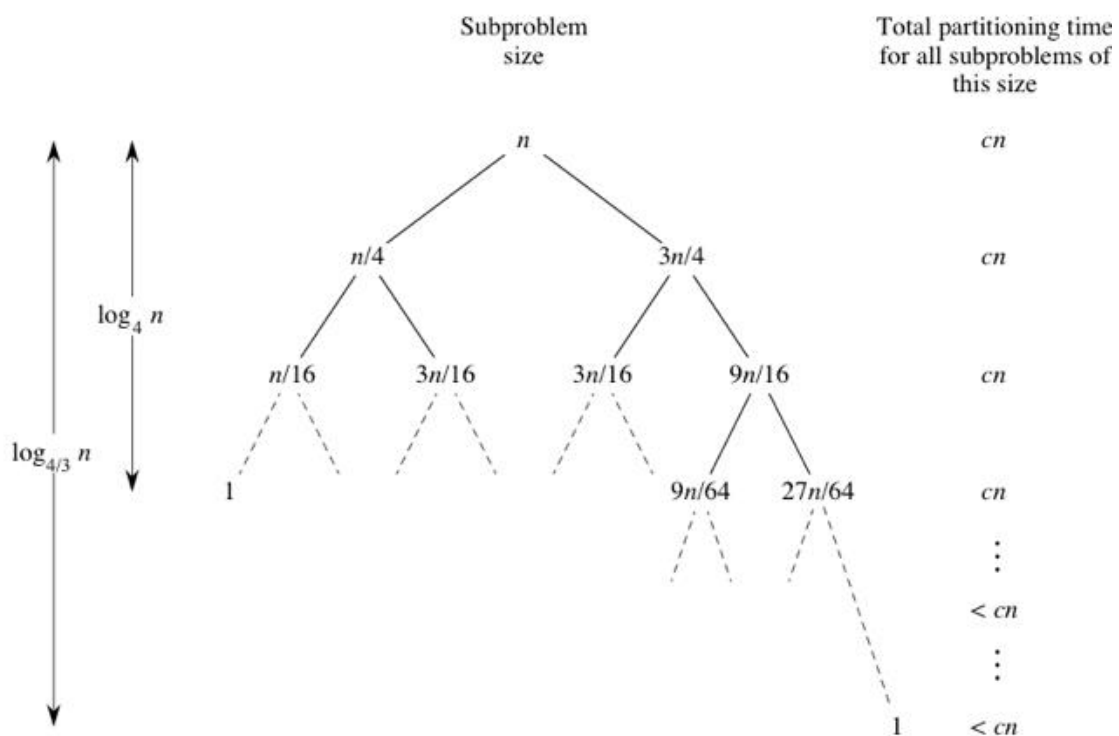
Sabiendo que ambos tienen la misma clase de complejidad, eso no significa precisamente que ambos tengan el mismo tiempo de ejecución. Quicksort suele ser más rápido que mergesort, porque es más fácil codificar las operaciones. Pero en la gráfica 2.2 y 2.3 se ve que no se da en todos los casos, para elementos aleatorios y de muchos elementos. No debemos usar Quicksort, usaremos MergeSort.

Para los elementos mayores a 1000000 la diferencia entre quick sort merge sort y insert sort

Tendríamos que el orden de mejor tiempo de ejecución es primero con quick sort, luego merge sort y por último insert sort.

Quicksort es solo $O(n \log n)$ en promedio y en el peor de los casos es $O(n^2)$ y Mergesort es siempre $O(n \log n)$.

CASO PROMEDIO



Caso promedio es $O(n \lg n)$

FIBONACCI

- I. Discuta sobre qué pasaría si intentáramos encontrar la respuesta en C++ y de forma recursiva? Qué pasaría si lo hiciéramos de forma iterativa?

Ventajas de la recursividad:

- Soluciones simples y claras
- Soluciones elegantes
- Soluciones a problemas complejos

Desventajas de la recursividad: INEFICIENCIA

- Sobrecarga asociada a las llamadas recursivas:

Una simple llamada puede generar un gran número de llamadas recursivas. (Fact(n) genera n llamadas recursivas)

¿La claridad compensa la sobrecarga?

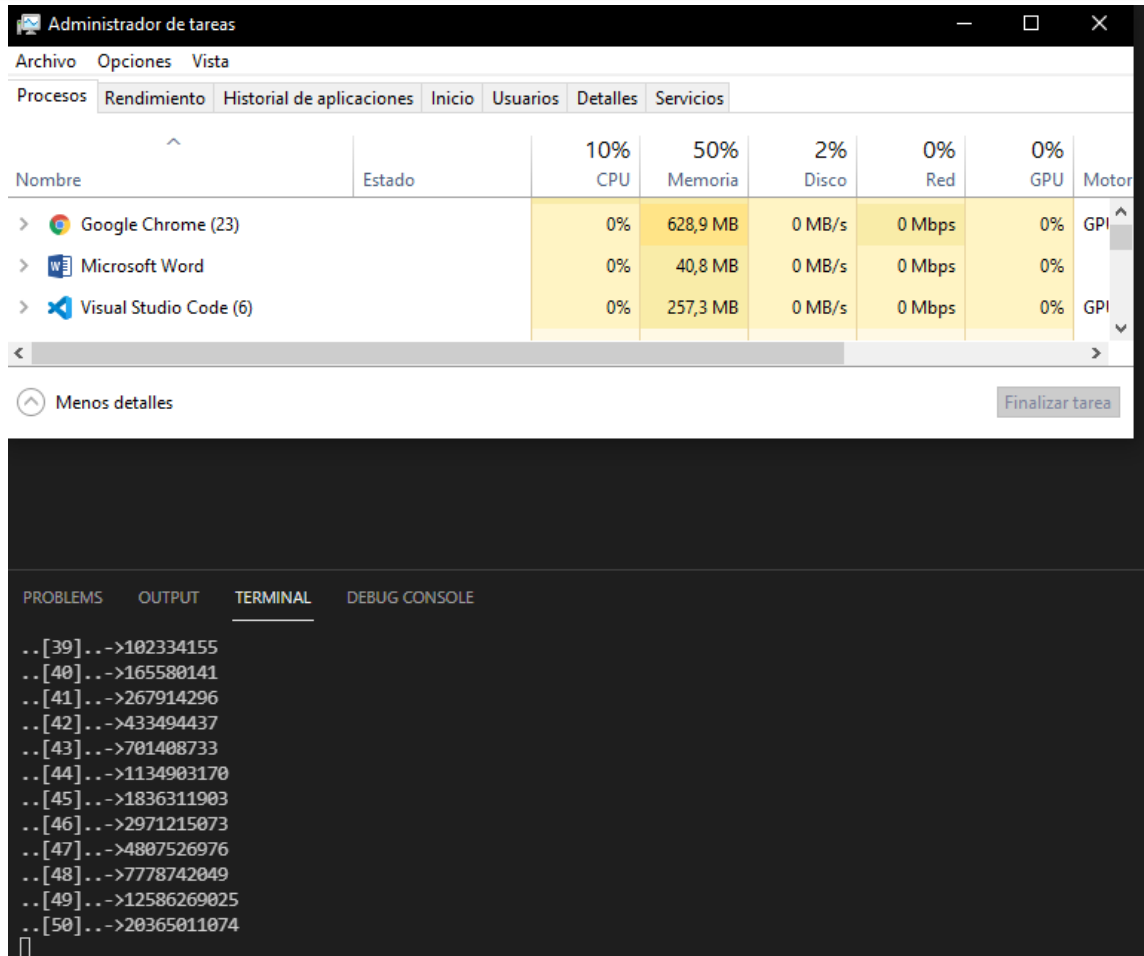
El valor de la recursividad reside en el hecho de que se puede usar para resolver problemas sin fácil solución iterativa.

- La ineficiencia inherente de algunos algoritmos recursivos.

La recursividad se debe usar cuando sea realmente necesaria, es decir, cuando no exista una solución iterativa simple.

Administrador de tareas						
Archivo Opciones Vista						
Procesos Rendimiento Historial de aplicaciones Inicio Usuarios Detalles Servicios						
Nombre	Estado	25% CPU	50% Memoria	11% Disco	0% Red	0% GPU
Aplicaciones (3)						
> Administrador de tareas		0,1%	22,8 MB	0 MB/s	0 Mbps	0%
> Google Chrome (23)		8,0%	668,2 MB	0,6 MB/s	0,3 Mbps	0%
> Visual Studio Code (6)		0%	196,1 MB	0,1 MB/s	0 Mbps	0%

RECURSIVO



The screenshot shows the Windows Task Manager interface. The 'Procesos' tab is active, displaying a list of running processes with columns for Name, State, CPU, Memory, Disk, Network, GPU, and Motor. The processes listed are Google Chrome (23), Microsoft Word, and Visual Studio Code (6). Below the task manager, a terminal window is open, showing a series of output lines from a program, likely a Fibonacci sequence generator, with values ranging from 102334155 to 20365011074.

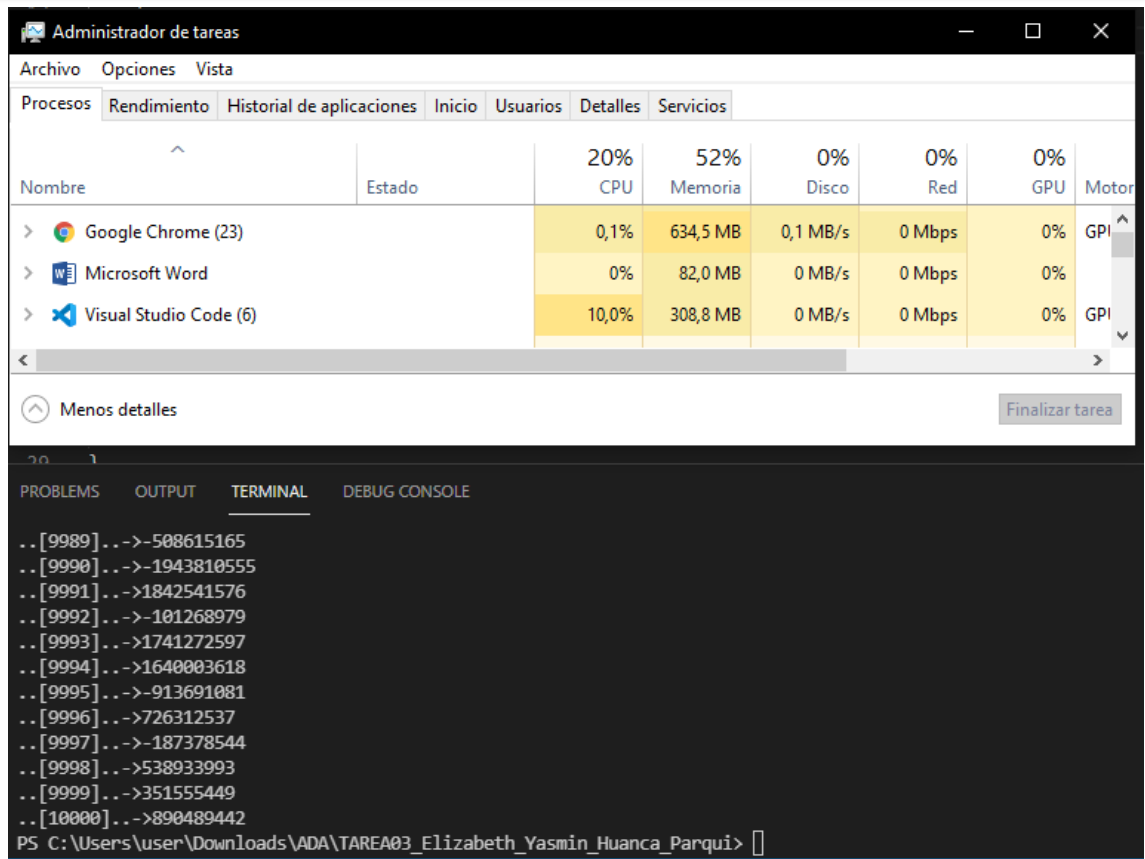
Nombre	Estado	10% CPU	50% Memoria	2% Disco	0% Red	0% GPU	Motor
Google Chrome (23)		0%	628,9 MB	0 MB/s	0 Mbps	0%	GPI
Microsoft Word		0%	40,8 MB	0 MB/s	0 Mbps	0%	
Visual Studio Code (6)		0%	257,3 MB	0 MB/s	0 Mbps	0%	GPI

```

PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE
..[39]..->102334155
..[40]..->165580141
..[41]..->267914296
..[42]..->433494437
..[43]..->701408733
..[44]..->1134903170
..[45]..->1836311903
..[46]..->2971215073
..[47]..->4807526976
..[48]..->7778742049
..[49]..->12586269025
..[50]..->20365011074
  
```

Demoro mucho para llegar al 10000, tomo mucho tiempo para llegar al Fibonacci 50. Este algoritmo tiene una solución iterativa simple por ende la recursividad no me sirve para resolver el problema.

Iterativo



The screenshot shows the Windows Task Manager window with the 'Procesos' tab selected. It displays the following data:

Nombre	Estado	20% CPU	52% Memoria	0% Disco	0% Red	0% GPU	Motor
Google Chrome (23)		0,1%	634,5 MB	0,1 MB/s	0 Mbps	0%	GPI
Microsoft Word		0%	82,0 MB	0 MB/s	0 Mbps	0%	
Visual Studio Code (6)		10,0%	308,8 MB	0 MB/s	0 Mbps	0%	GPI

Below the Task Manager window, the Visual Studio Code terminal is open, showing the output of a program calculating the Fibonacci sequence up to the 10000th term. The output is as follows:

```

..[9989]..->-508615165
..[9990]..->-1943810555
..[9991]..->1842541576
..[9992]..->-101268979
..[9993]..->1741272597
..[9994]..->1640003618
..[9995]..->-913691081
..[9996]..->726312537
..[9997]..->-187378544
..[9998]..->538933993
..[9999]..->351555449
..[10000]..->890489442
PS C:\Users\user\Downloads\ADA\TAREA03_Elizabeth_Yasmin_Huanca_Parqui>

```

Para 10000 me mando la respuesta rápidamente.
Se debe a que la solución iterativa de fibonacci es simple.

2. ¿Qué pasaría si tratásemos con doubles?

```

Agrega limite->10000
..[1]..->0
..[2]..->1
..[3]..->1
..[4]..->2
..[5]..->3
..[6]..->5
..[7]..->8
..[8]..->13
..[9]..->21
..[10]..->34
..[11]..->55
..[12]..->89
..[13]..->144
..[14]..->233
..[15]..->377
..[16]..->610
..[17]..->987
..[18]..->1597
..[19]..->2584

```

```

.[27]..->121393
.[28]..->196418
.[29]..->317811
.[30]..->514229
.[31]..->832040
.[32]..->1.34627e+006
.[33]..->2.17831e+006
.[34]..->3.52458e+006
.[35]..->5.70289e+006
.[36]..->9.22746e+006
.[37]..->1.49304e+007
.[38]..->2.41578e+007

```

Aquí se observa un desborde del `double`, nos está mandando desde el fibonacci 31 puras direcciones de memorias, "información basura".

```

..[9988]..->inf
..[9989]..->inf
..[9990]..->inf
..[9991]..->inf
..[9992]..->inf
..[9993]..->inf
..[9994]..->inf
..[9995]..->inf
..[9996]..->inf
..[9997]..->inf
..[9998]..->inf
..[9999]..->inf
..[10000]..->inf

```

3. ¿Qué ocurriría con la memoria si lo intentáramos en python?

Administrador de tareas

Archivo Opciones Vista

Procesos Rendimiento Historial de aplicaciones Inicio Usuarios Detalles Servicios

Nombre	Estado	3% CPU	50% Memoria	1% Disco	0% Red	1% GPU
> Google Chrome (23)		0,1%	480,6 MB	0,1 MB/s	0 Mbps	0%
> Microsoft Word		0%	85,1 MB	0 MB/s	0 Mbps	0%
> Python (2)		0,2%	9,6 MB	0 MB/s	0 Mbps	0%
> Visual Studio Code (14)		0%	332,7 MB	0 MB/s	0 Mbps	0%
Procesos en segundo plano (67)						
> Adaptador de rendimiento inver...		0%	0,9 MB	0 MB/s	0 Mbps	0%
> Adobe Acrobat Update Service (...)		0%	0,3 MB	0 MB/s	0 Mbps	0%

Para 10000 elementos-> empezamos con la memoria que se ve en la figura anterior.

Administrador de tareas

Archivo Opciones Vista

Procesos Rendimiento Historial de aplicaciones Inicio Usuarios Detalles Servicios

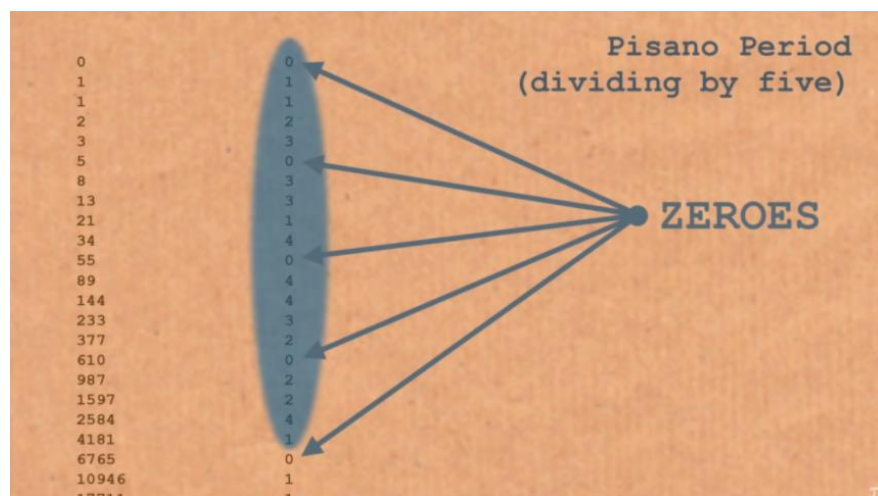
Nombre	Estado	26% CPU	50% Memoria	0% Disco	0% Red	15% GPU	Motor
> Google Chrome (23)		0,4%	482,1 MB	0,1 MB/s	0,1 Mbps	0%	GPI
> Microsoft Word		0%	84,9 MB	0 MB/s	0 Mbps	0%	
> Python (32 bits) (3)		21,8%	10,6 MB	0 MB/s	0 Mbps	0%	
> Visual Studio Code (14)		0%	330,7 MB	0 MB/s	0 Mbps	0%	GPI
Procesos en segundo plano (68)							
> Adaptador de rendimiento inver...		0,1%	0,9 MB	0 MB/s	0 Mbps	0%	
> Adobe Acrobat Update Service (...)		0%	0,3 MB	0 MB/s	0 Mbps	0%	

Menos detalles Finalizar tarea

Luego de la ejecución y al terminar se nota un aumento de CPU y de memoria.

4. ¿Cómo sería útil si utilizáramos la teoría de Aritmética Modular?

La Aritmética modular nos dio otra sucesión adicional a la de Fibonacci, con diferentes propiedades como por ejemplo la de Pisano, que al hallar los módulos de los Fibonacci obtuvimos nuevas sucesiones. Como por ejemplo



$$F_n | F_m \text{ if and only if } n | m$$

5. ¿Cómo sería si utilizáramos la forma matricial de Fibonacci junto a la forma de elevar un número a potencia de n pero utilizando una forma divide y vencerás? - Si, a todo esto, utilice la teoría de aritmética modular.

Generaremos cada matriz y cada operación para hallar el fibonacci

CUADRADOS

$$\text{número de cuadrados} = \frac{n(n+1)(2n+1)}{6}$$

3. CONCLUSIÓN

INSERT VS QUICKSORT VS MERGESORT

Tomando en cuenta la cantidad de elementos de cada arreglo llegamos a concluir que Insert Sort será el mejor para una pequeña cantidad de elementos, en cambio para las pruebas de MergeSort y QuickSort con elementos aleatorios y variando el tamaño de elementos deberemos utilizar MergeSort por conveniencia.

CASO PROMEDIO

En promedio para todos los elementos de entrada de tamaño n , el método hace $O(n \log n)$ comparaciones, el cual es relativamente eficiente

FIBONACCI

La recursividad es una técnica de programación muy conveniente para estudiar. Debemos de utilizarlo en algoritmos con iteración difícil.

CUADRADOS

Las fórmulas de cuadrados se obtiene por el análisis de partes para obtener la formula general

4. REFERENCIAS

1. <https://es.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quick-sort>
2. https://es.wikipedia.org/wiki/Aritm%C3%A9tica_modular
3. https://es.wikipedia.org/wiki/Sucesi%C3%B3n_de_Fibonacci

5. LINKS DE AVANCE



<https://github.com/ElizabethYasmin/ADA-Trabajos>

<https://docs.google.com/document/d/10jhjVxVj4QYrkf48BL73AWelCvXwC8Yp2ZCS8A6oPOM/edit#heading=h.38sy7ba0ptiq>