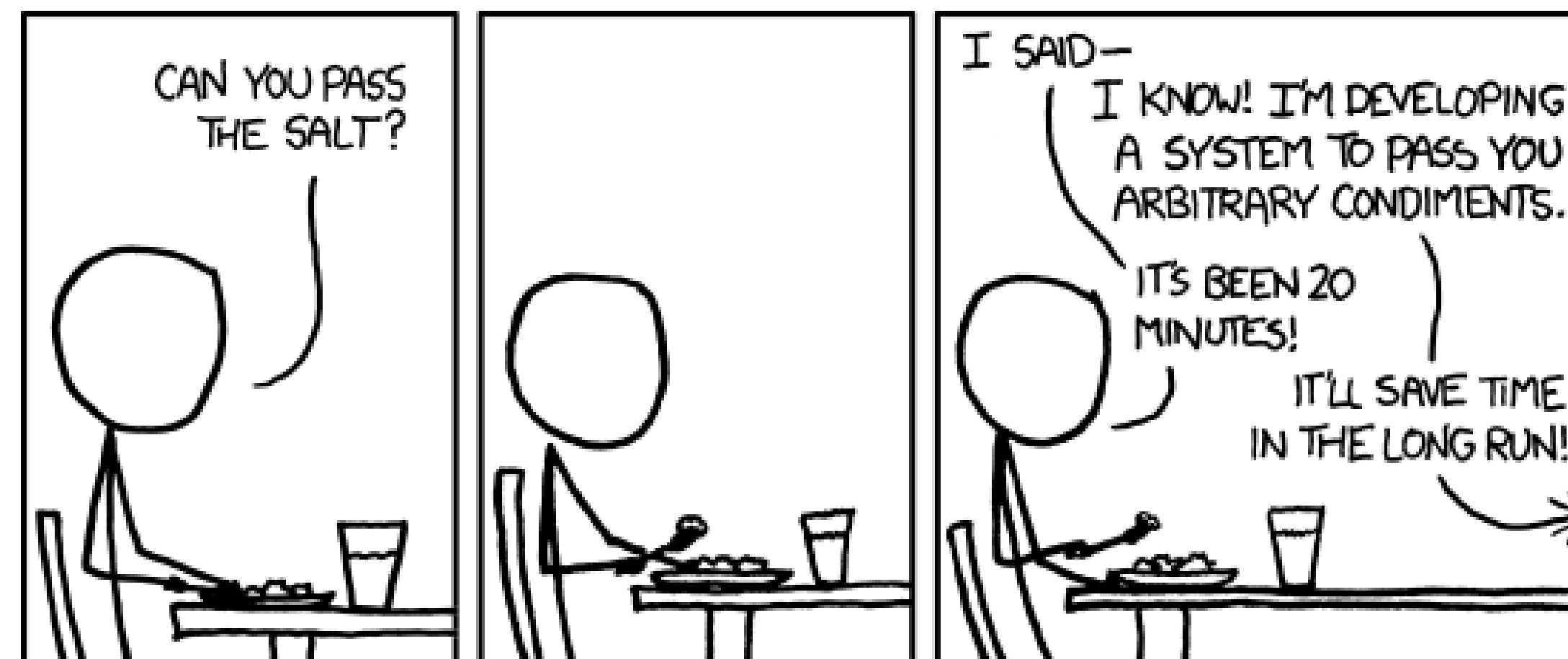
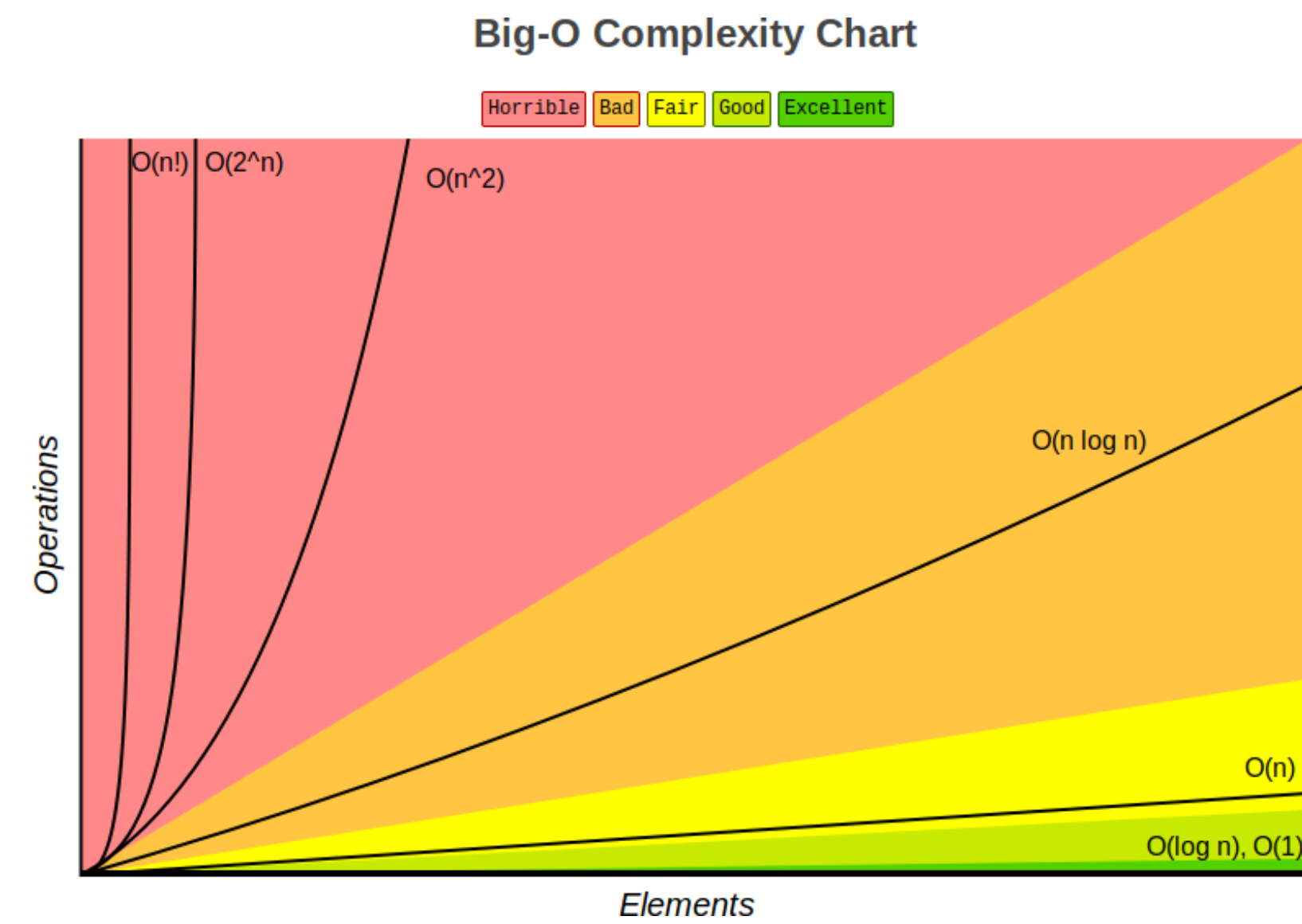


# Optimización, paralelismo, concurrencia y cómputo distribuido en alto nivel.



# Complejidad

O: que límite de tiempo no supera un algoritmo al infinito

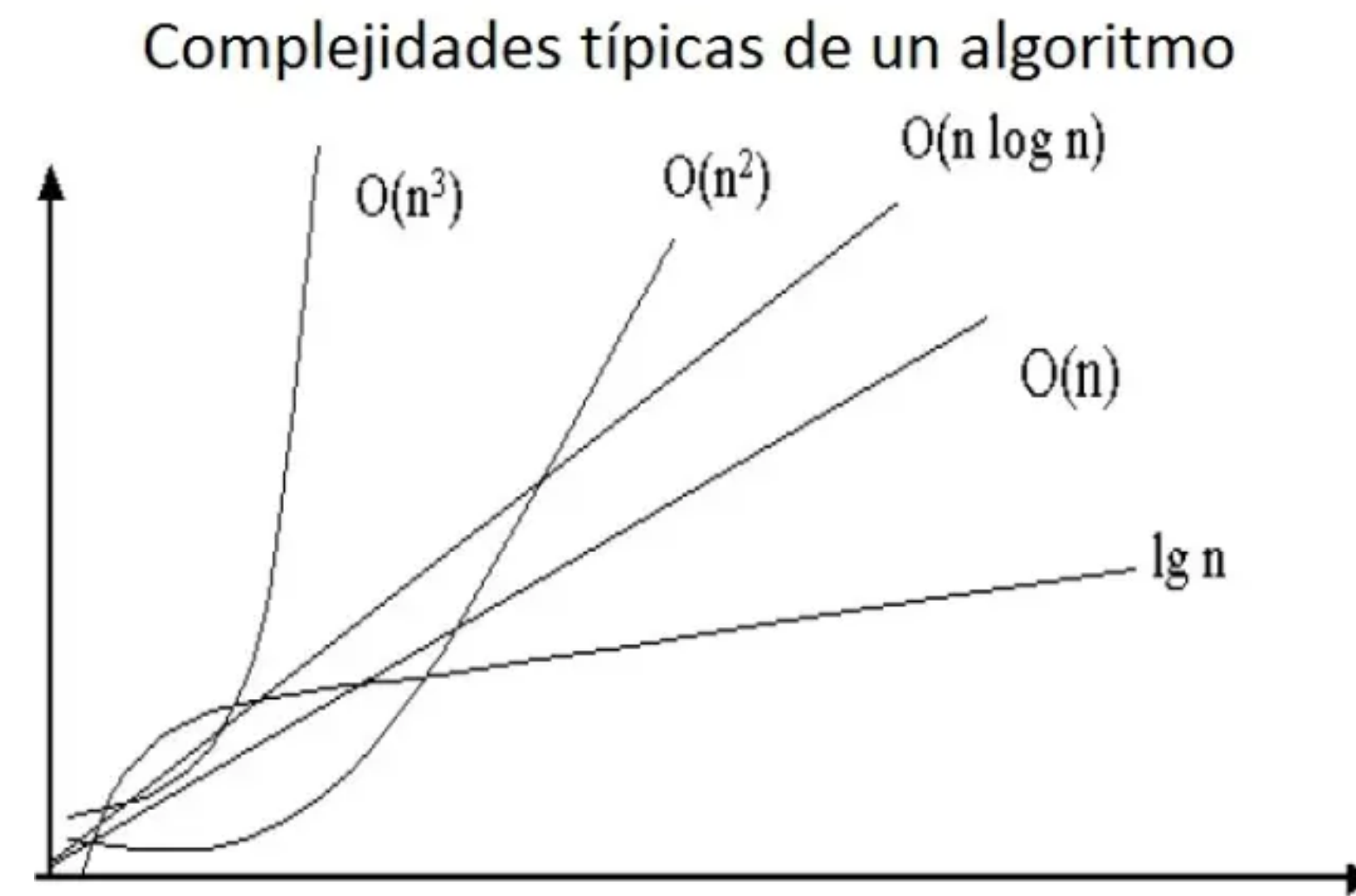


Algoritmos	Orden de Complejidad
Quicksort	$O(n \log n)$
Bubble-sort	$O(n^2)$
Shell-sort	$O(n^{1.25})$
Heap-sort	$O(n \log n)$
Inserción	$O(n^2)$
Selección	$O(n^2)$
Merge-Sort	$O(n \log n)$

- <https://www.youtube.com/watch?v=ZZuD6iUe3Pc>

# Complejidad la realidad -

En pocos valores o en casos particulares la complejidad no es lo mismo



Array Sorting Algorithms				
Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(\log(n))$
Mergesort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$
Timsort	$\Omega(n)$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$
Heapsort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(1)$
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Tree Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(n)$
Shell Sort	$\Omega(n \log(n))$	$\Theta(n(\log(n))^2)$	$O(n(\log(n))^2)$	$O(1)$
Bucket Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n^2)$	$O(n)$
Radix Sort	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$	$O(n+k)$
Counting Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n+k)$	$O(k)$
Cubesort	$\Omega(n)$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$

Un detalle es que **Ingeniería de Software** la complejidad se refiere a casos promedios y no límites como es en CS.

# Optimización de programas

*El proceso de modificación de un sistema de software para que algún aspecto funcione de manera más eficiente o use menos recursos.*

- Un programa de computadora puede optimizarse para que se ejecute más rápidamente, para que sea capaz de operar con menos almacenamiento, memoria u otros recursos.
- No existe un diseño de "talla única" que funcione bien en todos los casos, por lo que se hacen concesiones para optimizar los atributos de mayor interés.
- Optimizar suele venir en detrimento de muchos indicadores de calidad (como legibilidad).
- **NO SE OPTIMIZA SIN TESTS.**

# Strength reduction

- En la construcción del compilador, la **reducción de la resistencia** es una optimización del compilador donde las operaciones costosas se reemplazan por operaciones equivalentes pero menos costosas.
- El ejemplo clásico de reducción de fuerza convierte las multiplicaciones "fuertes" dentro de un bucle en adiciones "más débiles".

```
c = 7;  
for (i = 0; i < N; i++) {  
    y[i] = c * i;  
}
```

Puede ser reemplazado por lo siguiente:

```
c = 7;  
k = 0;  
for (i = 0; i < N; i++) {  
    y[i] = k;  
    k = k + c;  
}
```

# Strength reduction

- Fragmento de código C cuya intención es obtener la suma de todos los enteros de 1 a N (No se asume overflows):

```
int i, sum = 0;
for (i = 1; i <= N; ++i) {
    sum += i;
}
printf("sum: %d\n", sum);
```

Puede ser reemplazado por lo siguiente:

```
int sum = N * (1 + N) / 2;
printf("sum: %d\n", sum);
```

# Niveles de optimización

- La optimización puede ocurrir en varios niveles.
  - Los niveles más altos tienen un mayor impacto y son más difíciles de cambiar más adelante en un proyecto (API).
- La optimización generalmente se realiza **de mayor a menor**, con ganancias iniciales mayores y logradas con menos trabajo, y ganancias posteriores más pequeñas y que requieren más trabajo.
- En algunos casos, el rendimiento general depende del rendimiento de porciones de muy bajo nivel conceptual de un programa, y los pequeños cambios en una etapa tardía o la consideración temprana de los detalles de bajo nivel pueden tener un impacto descomunal.

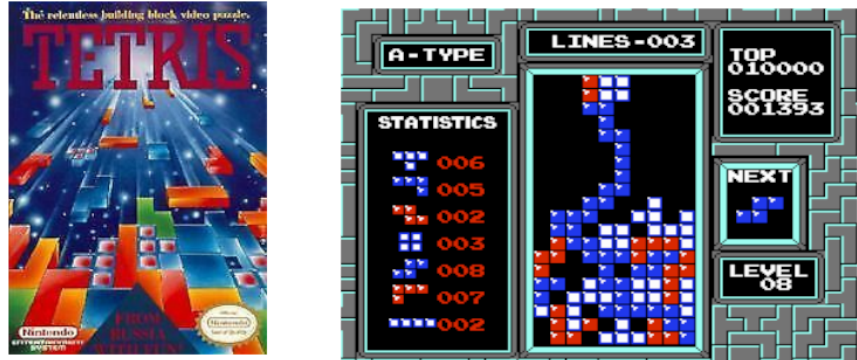
## Niveles de optimización - Design level

- El diseño arquitectónico de un sistema afecta abrumadoramente su rendimiento. Un sistema que está vinculado a la latencia de la red se optimizaría para minimizar los viajes de red.
- La elección del diseño depende de los objetivos.
- La elección de la plataforma y el lenguaje de programación se producen en este nivel, y cambiarlos con frecuencia requiere una reescritura completa, aunque un sistema modular puede permitir la reescritura de solo algunos componentes; por ejemplo, un programa Python puede reescribir secciones críticas para el rendimiento en C.



# Optimización pensada desde el alto nivel.

Tres rutinas de random en juegos implementados en Assembler de 6502 del Famicom.



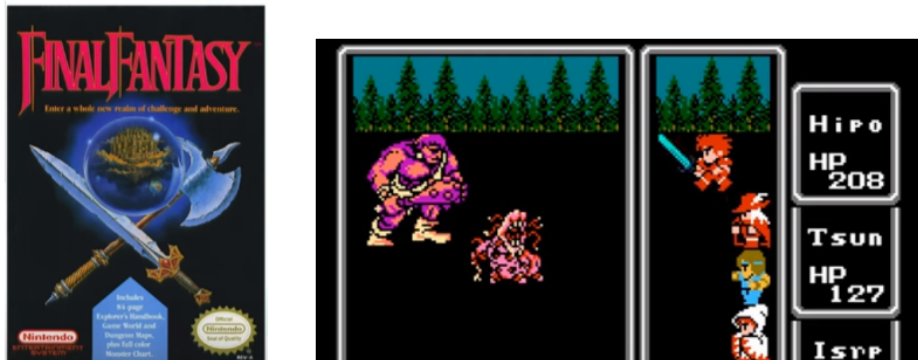
THE TETRIS SOLUTION: DO IT WITH MATH

16-bit Fibonacci linear feedback shift register (LFSR)



(XOR bits 1 and 9, store in bit 16, shift right)

[https://meatfighter.com/nintendotetrisai/#Picking\\_Tetriminos](https://meatfighter.com/nintendotetrisai/#Picking_Tetriminos)



THE FINAL FANTASY SOLUTION: USE A LOOKUP TABLE

AE	D0	38	8A	ED	60	DB	72	5C	59	27	D8	0A	4A	F4	34	08	A9	C3	96
56	3B	F1	55	F8	6B	31	EF	6D	28	AC	41	68	1E	2A	C1	E5	8F	50	F5
3E	7B	B7	4C	14	39	12	CD	B2	62	8B	82	3C	BA	63	85	3A	17	B8	2E
B5	BE	20	CB	46	51	2C	CF	03	78	53	97	06	69	EB	77	86	E6	EA	74
0C	21	E2	40	D4	5A	3D	C7	2B	94	D5	8C	44	FD	EE	D2	43	00	BB	FA
C6	1D	98	A0	D3	54	5F	5E	DC	A8	00	AF	93	A1	E1	6C	04	DE	B6	D7
36	16	C5	C8	C4	E4	0F	02	AB	E8	33	99	73	11	6A	09	67	F3	FF	A2
DF	32	0E	1F	0D	90	25	64	75	B3	65	2F	C9	B0	DA	5D	9F	EC	29	CE
E3	F0	91	7A	58	45	24	1C	47	A4	89	18	2D	CC	BD	6F	80	F6	81	22
E9	07	70	FB	DD	AD	35	A6	61	B4	A3	FE	B1	30	4B	15	48	6E	4F	5B
13	9C	83	92	01	C2	19	7F	1A	1B	71	B9	3F	4E	9B	BF	9E	87	0B	10
57	F2	26	79	9A	05	C0	E0	F7	4D	7D	CA	52	9D	F9	BC	AA	FC	8D	7E
D1	A5	42	E7	D6	76	A7	84	8E	66	7C	23	88	37	49	D9				



THE CONTRA SOLUTION: DO SOME REALLY FAST MATH, ALL THE TIME

“Contra has a single global 8-bit value that it uses as the source of randomness throughout the game. ...[T]he next random value is generated by spinning in a tight loop during the time that the game is idle and waiting for the next display frame to begin.

-Allan Blomquist

<https://www.youtube.com/watch?v=TPbroUDHG0s>

## Niveles de optimización - Algorithms and data structures

- Después del diseño, la elección de algoritmos y estructuras de datos afecta la eficiencia más que cualquier otro aspecto del programa.
- En general, las estructuras de datos son más difíciles de cambiar que los algoritmos, ya que una suposición de estructura de datos y sus suposiciones de rendimiento se utilizan en todo el programa.
- Consiste principalmente en asegurar que los algoritmos sean  $O(1)$  constante,  $O(\log n)$  logarítmico,  $O(n)$  lineal o, en algunos casos, log lineal  $O(n \log n)$  tanto en el espacio como en tiempo).
- Los algoritmos con complejidad cuadrática  $O(n^2)$  no escalan, e incluso los algoritmos lineales causan problemas si se llaman repetidamente.
- Una técnica general para mejorar el rendimiento es evitar el trabajo.
  - Por ejemplo es el uso de *fast-path* para casos comunes.
  - Otra técnica es caché, en particular *memoization* que evita cálculos redundantes.

# Si cambiamos una estructura de datos de alto nivel?

```
In [1]: import pandas as pd; import numpy as np

class DF2Pandas:
    def __init__(self, c0, c1):
        self._df = pd.DataFrame({"c0": c0, "c1": c1})
    def mean(self):
        return self._df.mean()

class DF2Numpy:
    def __init__(self, c0, c1):
        self.c0, self.c1 = np.asarray(c0), np.asarray(c1)
    def mean(self):
        return pd.Series({"c0": np.mean(self.c0), "c1": np.mean(self.c1)})

c0, c1 = np.random.rand(2, 1000)
```

# Si cambiamos una estructura de datos de alto nivel?

In [4]: `%timeit DF2Pandas(c0, c1).mean()`

207  $\mu\text{s}$   $\pm$  3.7  $\mu\text{s}$  per loop (mean  $\pm$  std. dev. of 7 runs, 1,000 loops each)

In [5]: `%timeit DF2Numpy(c0, c1).mean()`

91.1  $\mu\text{s}$   $\pm$  2.44  $\mu\text{s}$  per loop (mean  $\pm$  std. dev. of 7 runs, 10,000 loops each)



## Niveles de optimización - Source code

- Las elecciones concretas de nivel de código fuente pueden marcar una diferencia significativa.
- Por ejemplo, en Python **las listas por comprension suelen ser mas rapidas que los *for-loops*.**
- Algunas optimizaciones (como esta) se pueden realizar hoy en día mediante la optimización de compiladores. -
- Este es un lugar clave donde la comprensión de los compiladores y el código de la máquina puede mejorar el rendimiento.
- El movimiento de código invariante fuera de un bucle y la optimización del valor de retorno son ejemplos de optimizaciones que reducen la necesidad de variables auxiliares e incluso pueden dar como resultado un rendimiento más rápido al evitar optimizaciones redondas.
- Más ejemplos: <https://stackabuse.com/python-performance-optimization/>

# Mejoremos una operación tonta

```
In [7]: mtx = np.random.rand(50, 500)
```

```
In [12]: %%timeit
result = []
for r in mtx:
    nrow = []
    for c in r:
        nrow.append(c / r.sum())
    result.append(np.sum(nrow))
np.array(result)
```

26.2 ms  $\pm$  377  $\mu$ s per loop (mean  $\pm$  std. dev. of 7 runs, 10 loops each)

# Mejoremos una operación tonta

```
In [9]: %%timeit
result = []
for r in mtx:
    nrow = []
    total = r.sum() # no haga la suma en cada iteracion
    for c in r:
        nrow.append(c/total)
    result.append(np.sum(nrow))
np.array(result)
```

2.43 ms  $\pm$  26  $\mu$ s per loop (mean  $\pm$  std. dev. of 7 runs, 100 loops each)

## Mejoremos una operación tonta

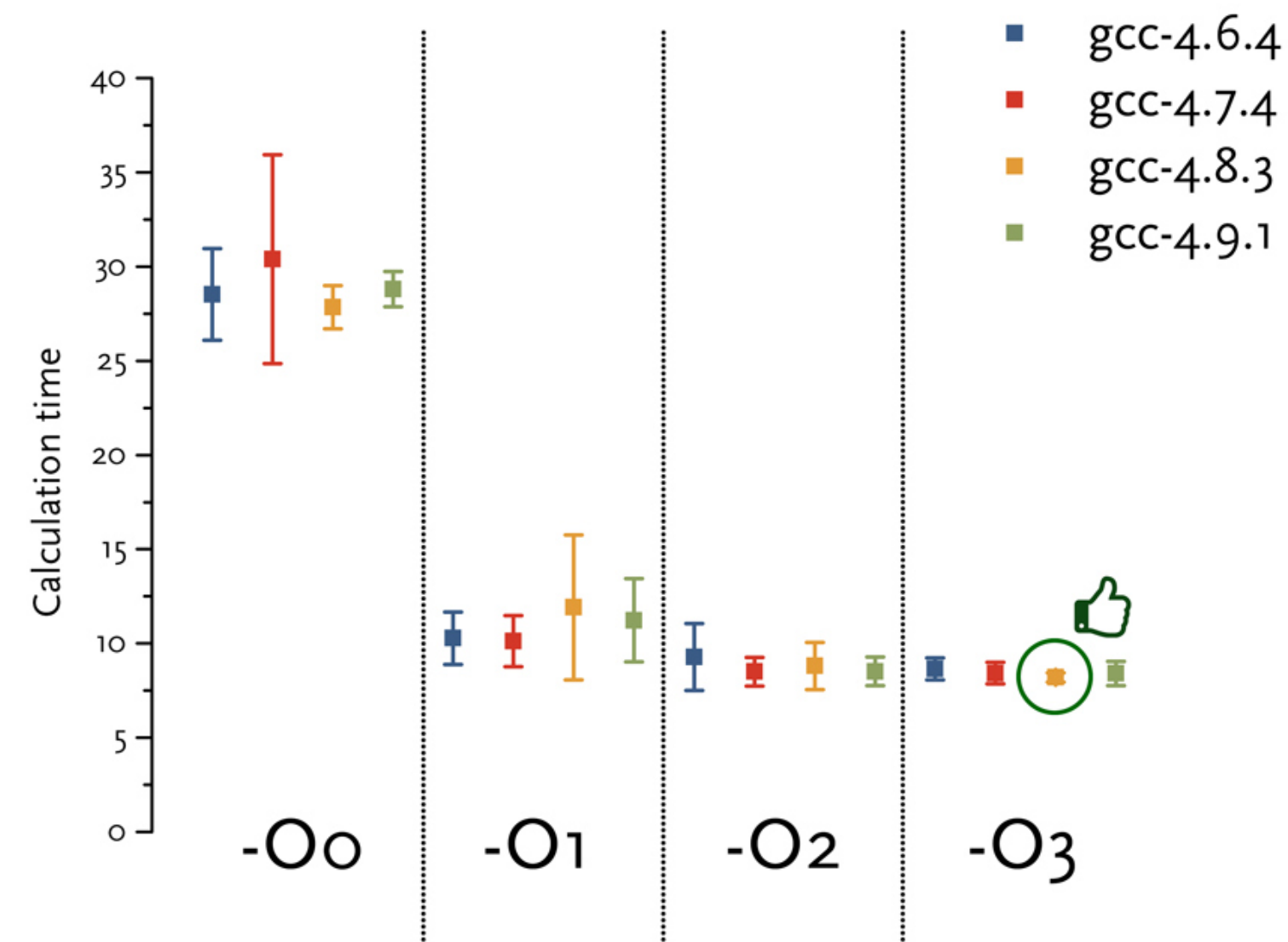
```
In [13]: %%timeit  
(mtx / mtx.sum(axis=1, keepdims=True)).sum(axis=1)
```

29  $\mu$ s  $\pm$  343 ns per loop (mean  $\pm$  std. dev. of 7 runs, 10,000 loops each)



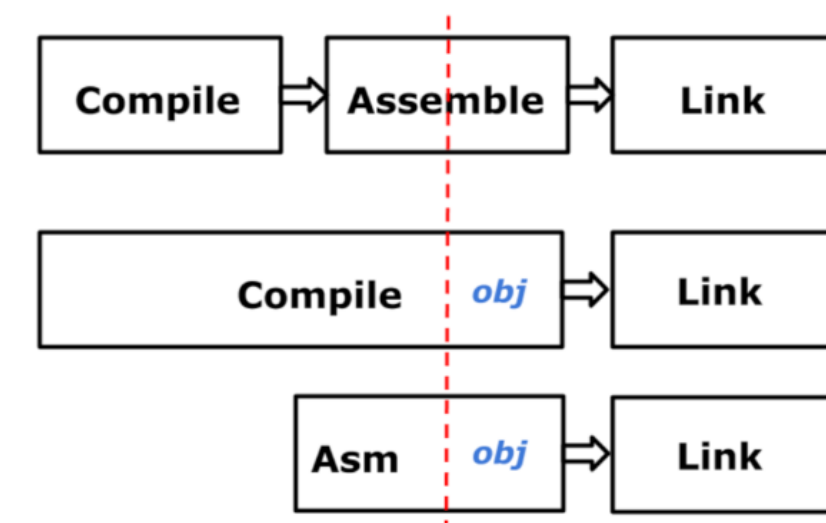
## Niveles de optimización - Build level

- Los flags de compilación ajustan las opciones de rendimiento en el código fuente y el compilador respectivamente,
- También puede deshabilitarse las funciones de software innecesarias.
- Optimización para modelos de procesador específicos o capacidades de hardware, o la predicción de ramificaciones, por ejemplo.



# Niveles de optimización - Assembly level

- En el nivel más bajo, escribir código usando un lenguaje ensamblador, diseñado para una plataforma de hardware particular puede producir el código más eficiente y compacto si el programador aprovecha el repertorio completo de instrucciones de la máquina.
- Dado los compiladores modernos y la mayor complejidad de las CPU recientes, es más difícil escribir código más eficiente que el que genera el compilador, y pocos proyectos necesitan este paso de optimización "definitivo".
- Normalmente, en lugar de escribir en lenguaje ensamblador, los programadores usarán un desensamblador para analizar la salida de un compilador y cambiar el código fuente de alto nivel para que pueda compilarse de manera más eficiente, o entiendan por qué es ineficiente.



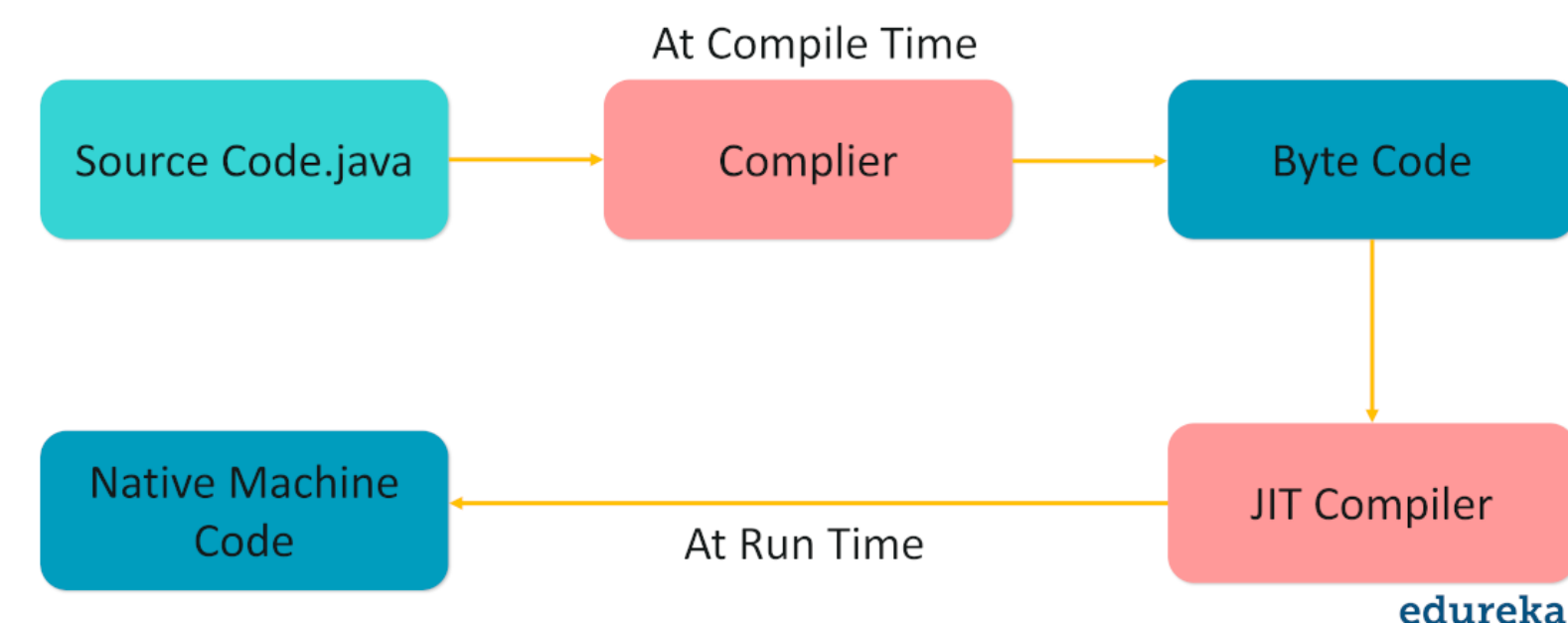
## Niveles de optimización - Run-Time

- Los compiladores justo a tiempo (JIT) pueden producir código de máquina personalizado basado en datos de tiempo de ejecución, a costa de la sobrecarga de la compilación.
- En algunos casos se puede realizar una optimización que excede la capacidad de los compiladores estáticos.
- La optimización guiada por profiling es una técnica de optimización de compilación anticipada (AOT) basada en perfiles de tiempo de ejecución, y es similar a un análogo estático de "caso promedio" de la técnica dinámica de optimización adaptativa.
- Algunos diseños de CPU pueden realizar algunas optimizaciones en tiempo de ejecución. Algunos ejemplos incluyen ejecución fuera de orden, ejecución especulativa, canalizaciones de instrucciones y predictores de rama.

 Numba

 julia

 pypy



# Compensaciones

- La optimización se basa en el uso de algoritmos más elaborados, haciendo uso de "casos especiales" realizando complejas compensaciones.
- Un programa "optimizado" puede ser más difícil de comprender y, por lo tanto, puede contener más fallas que las versiones no optimizadas.
- Algunas optimizaciones de nivel de código disminuyen la capacidad de mantenimiento.
- La optimización generalmente se centrará en mejorar solo algunos aspectos:
  - tiempo de ejecución.
  - uso de memoria.
  - espacio en disco.
  - ancho de banda.
  - o algún otro recurso.
- **Esto generalmente implica que un factor se optimice a expensas de otros.**
- Por ejemplo, más memoria caché mejora el rendimiento del tiempo de ejecución, pero también aumenta el consumo de memoria.

# Compensaciones

- Hay casos en los que el programador que realiza la optimización debe decidir mejorar el software para algunas operaciones, pero a costa de hacer que otras operaciones sean menos eficientes.
- Estas compensaciones a veces pueden ser de naturaleza no técnica, como cuando un competidor ha publicado un resultado de referencia que debe superarse para mejorar el éxito comercial, pero tal vez conlleva que el uso normal del software sea menos eficiente.
- Tales cambios a veces se denominan en broma **pessimizations**.

# Bottlenecks

- Un cuello de botella en un sistema, es un componente que es el factor limitante en el rendimiento.
- En ciencias de la computación, el consumo de recursos a menudo sigue una forma de distribución de power-law, y el principio de Pareto se puede aplicar a la optimización de recursos.
- En ingeniería de software, a menudo es una mejor aproximación que el 90% del tiempo de ejecución de un programa de computadora se gasta ejecutando el 10% del código (conocido en este contexto como la ley 90/10).
- Los algoritmos más complejos y las estructuras de datos funcionan bien con muchos elementos, mientras que los algoritmos simples son más adecuados para pequeñas cantidades de datos
- El tiempo de inicialización y los factores constantes del algoritmo más complejo pueden superar su beneficio
- Un algoritmo híbrido o adaptativo suele ser la solución.
- En algunos casos, agregar más memoria puede ayudar a que un programa se ejecute más rápido.

Cuando optimizar



# Cuando optimizar

- Nunca





# Cuando optimizar

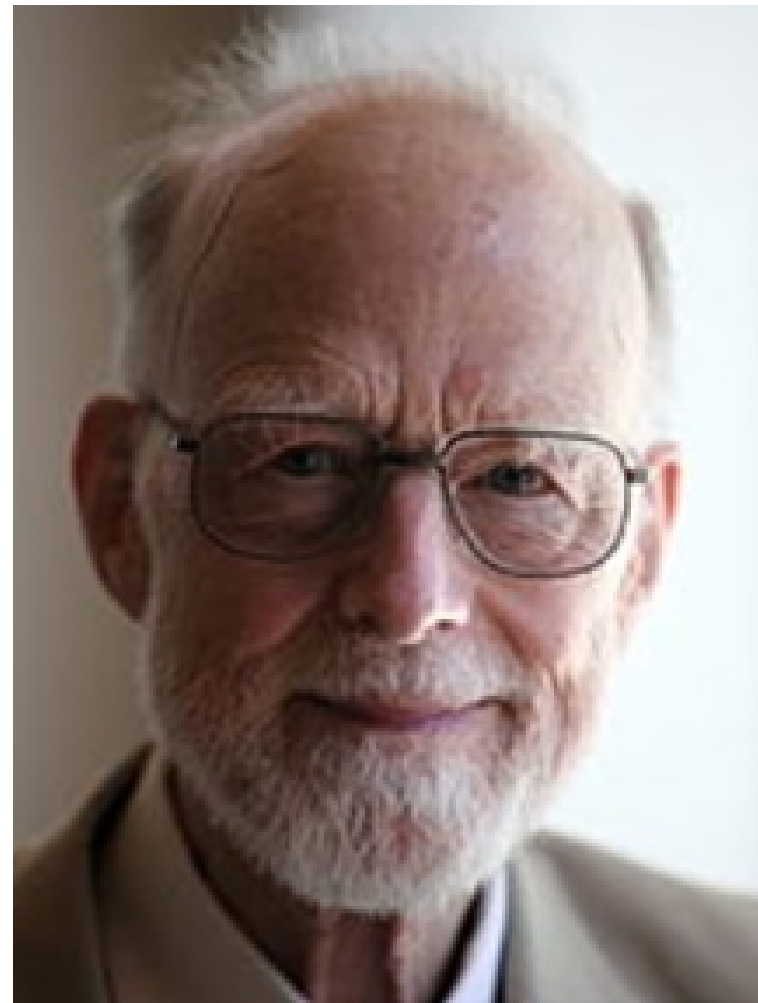
- Nunca
- No aún



# Cuando optimizar

- Nunca
- No aún
- Posta: NUNCA





There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.

— Tony Hoare —

AZ QUOTES



# Cuando optimizar

Donald Knuth (o Tony Hoare) hizo las siguientes dos declaraciones sobre optimización:

*Deberíamos olvidarnos de las pequeñas eficiencias, digamos alrededor del 97% del tiempo: la optimización prematura es la raíz de todo mal. Sin embargo, no debemos dejar pasar nuestras oportunidades en ese crítico 3%*

*En las disciplinas de ingeniería establecidas, una mejora del 12% obtenida fácilmente, nunca se considera marginal y creo que el mismo punto de vista debería prevalecer en el software"*

"Optimización prematura" es una frase utilizada para describir una situación en la que un programador permite que las consideraciones de rendimiento afecten el diseño de un fragmento de código.

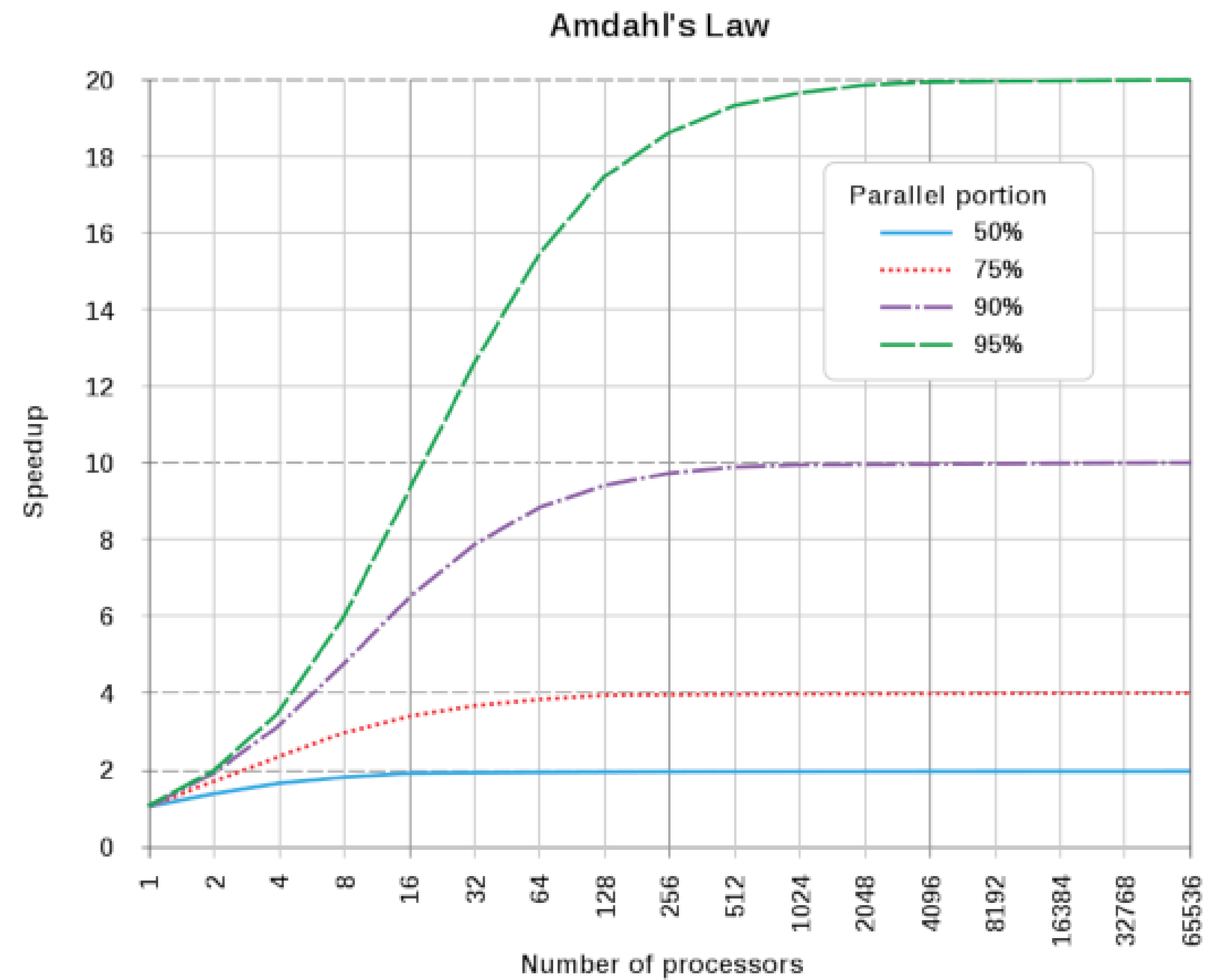
# Cuando optimizar

Al decidir si optimizar una parte específica del programa, siempre se debe considerar **la Ley de Amdahl**:

*el impacto en el programa general depende en gran medida de cuánto tiempo se dedica realmente a esa parte específica, lo que no siempre está claro al mirar el código sin un análisis de desempeño.*

- Es mejor diseñar primero, codificar desde el diseño y luego perfilar para ver qué partes deben optimizarse.
- Un diseño simple y elegante a menudo es más fácil de optimizar, y los perfiles puede revelar problemas de rendimiento inesperados.

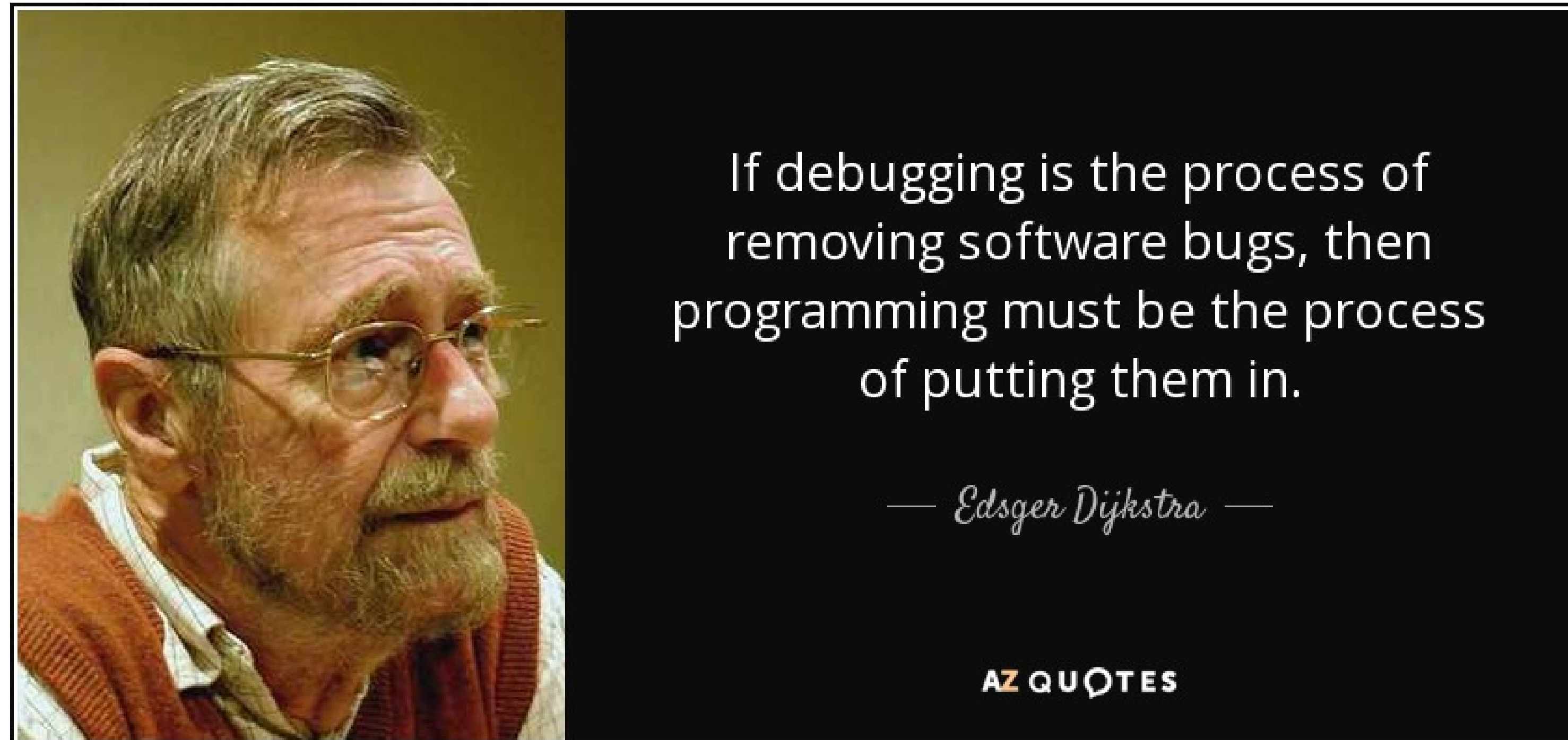
# Ley de Amdahl



# Tiempo necesario para la optimización.

- A veces, el tiempo necesario para llevar a cabo la optimización en sí mismo puede ser un problema.
- La optimización del código existente generalmente no agrega nuevas características y, lo que es peor, podría agregar nuevos errores en el código que funcionaba anteriormente.
- Debido a que el código optimizado manualmente a veces puede tener menos "legibilidad" que el código no optimizado, la optimización también puede afectar el mantenimiento del mismo.
- **La optimización tiene un precio y es importante asegurarse de que la inversión valga la pena.**

Bueno... todo esto al final es muy mala idea





# Referencias

- [https://en.wikipedia.org/wiki/Program\\_optimization](https://en.wikipedia.org/wiki/Program_optimization)
- [https://en.wikipedia.org/wiki/Strength\\_reduction](https://en.wikipedia.org/wiki/Strength_reduction)
- <https://stackabuse.com/python-performance-optimization/>