Algoritmos y estructuras de datos

Eficiencia de algoritmos

Francisco Javier Zaragoza Martínez

Universidad Autónoma Metropolitana Unidad Azcapotzalco
Departamento de Sistemas

15 de julio de 2024

Citas

Ley de Moore (1965)

La cantidad de transistores en un circuito integrado se duplica cada dos años.

Ley de Wirth (1995)

El software se alenta más rápidamente que lo que se acelera el hardware.

Ley de Parkinson (1955)

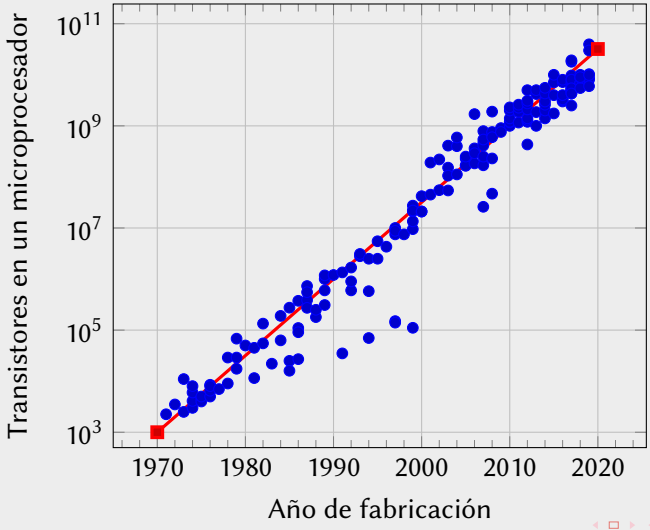
Los datos se expanden hasta llenar todo el espacio disponible.

Ley de May (2011)

La eficiencia del software se reduce a la mitad cada 18 meses.

50 años de la Ley de Moore

La cantidad de transistores en un integrado se duplica cada dos años



Eficacia y eficiencia

En ingeniería

La eficacia se refiere a que un sistema cumpla el objetivo para el que fué diseñado. La eficiencia se refiere a que además no desperdicie los recursos.

En computación

La eficacia se refiere a que un algoritmo o programa funcione siempre. En computación los recursos principales son el tiempo de ejecución y la memoria, así que la eficiencia se refiere a que además el algoritmo o programa haga pocos pasos y use poca memoria. Al estudio de estas propiedades se le llama el análisis de correctitud y de complejidad.

Relevancia

De ver la gráfica anterior uno podría pensar que si el hardware es cada vez más rápido esto no importa. Los que piensan así están muy equivocados.

Tiempo de ejecución de un algoritmo

El tamaño de la entrada

El tiempo que tarde un algoritmo en ejecutarse normalmente dependerá de la cantidad de datos que reciba a la entrada. Esa cantidad de datos se llama el tamaño de la entrada y usualmente la denotaremos por n (por ejemplo, la cantidad de bits de un entero, la cantidad de caracteres de una cadena o la cantidad de elementos de un arreglo).

El tiempo de ejecución

El tiempo de ejecución $T(n)$ de un algoritmo es el máximo tiempo que se tarde en ejecutarse si recibe una entrada de tamaño n (es decir, el peor caso si no siempre se tarda lo mismo con dos entradas del mismo tamaño). De manera práctica $T(n)$ se mediría en segundos pero, para no depender de la implementación, de manera teórica $T(n)$ se mide en cantidad de operaciones (por ejemplo, de bits, de caracteres o de enteros).

Ejemplos

Ejemplo (Paridad de un entero)

Para un entero de n bits se hace 1 operación $\Rightarrow T(n) = 1$.

Ejemplo (Búsqueda en un arreglo)

Para un arreglo de tamaño n se hacen $\leq n$ operaciones $\Rightarrow T(n) = n$.

Ejemplo (Comparación de dos cadenas)

Para dos cadenas de n caracteres se hacen $\leq n$ operaciones $\Rightarrow T(n) = n$.

Ejemplo (Comparación de dos arreglos desordenados)

Para dos arreglos de tamaño n se hacen $\leq n^2$ operaciones $\Rightarrow T(n) = n^2$.

Comparación de tiempos de ejecución típicos

Elegiremos una serie de ideas algorítmicas con tiempos de ejecución típicos. Estos algoritmos recibirán una entrada de tamaño n y tardarán $T(n)$ unidades de tiempo.

¿Qué pasa si duplicamos...

- el tamaño de la entrada? Es decir, el efecto de la Ley de Parkinson: $T(2n)$.
- la velocidad del procesador? Es decir, el efecto de la Ley de Moore: $T'(n)$.
- las dos cosas al mismo tiempo? Es decir, el efecto de la Ley de Wirth: $T'(2n)$.

Tiempo de ejecución lineal

Supón que $T(n) = n$. Esto pasa con `for (i = 0; i < n; i++)`

Duplicar la entrada

Nota que $T(2n) = 2n = 2T(n)$. Es decir, se tarda el doble que antes.

Duplicar la velocidad

Nota que $T'(n) = \frac{1}{2}T(n)$. Es decir, se tarda la mitad que antes.

Duplicar la entrada y duplicar la velocidad

Nota que $T'(2n) = \frac{1}{2}T(2n) = \frac{2}{2}T(n) = T(n)$. Es decir, se tarda lo mismo.

Tiempo de ejecución raíz cuadrático

Supón que $T(n) = \sqrt{n}$. Esto pasa con `for (i = 1; i*i <= n; i++)`

Duplicar la entrada

$T(2n) = \sqrt{2n} = \sqrt{2}\sqrt{n} = \sqrt{2}T(n)$. Se tarda $\approx 141\,\%$ que antes.

Duplicar la velocidad

$T'(n) = \frac{1}{2}T(n)$. Se tarda la mitad que antes.

Duplicar la entrada y duplicar la velocidad

$T'(2n) = \frac{1}{2}T(2n) = \frac{\sqrt{2}}{2}T(n) = \frac{1}{\sqrt{2}}T(n)$. Se tarda $\approx 71\,\%$ que antes.

Tiempo de ejecución logarítmico

Supón que $T(n) = \log_2 n$. Esto pasa con `for (i = 2; i <= n; i *= 2)`

Duplicar la entrada

$T(2n) = \log_2(2n) = T(n) + 1$. Se tarda un paso más que antes.

Duplicar la velocidad

$T'(n) = \frac{1}{2}T(n)$. Se tarda la mitad que antes.

Duplicar la entrada y duplicar la velocidad

$T'(2n) = \frac{1}{2}T(2n) = \frac{1}{2}(T(n) + 1)$. Se tarda $\approx 50\,\%$ que antes.

Tiempo de ejecución linearítmico

Supón que $T(n) = n \log_2 n$. Después veremos ejemplos de esto.

Duplicar la entrada

$T(2n) = (2n)\log_2(2n) = (2n)(\log_2 2 + \log_2 n) \approx 2(n \log_2 n) = 2T(n)$. Se tarda $\approx$ el doble que antes.

Duplicar la velocidad

$T'(n) = \frac{1}{2}T(n)$. Se tarda la mitad que antes.

Duplicar la entrada y duplicar la velocidad

$T'(2n) = \frac{1}{2}T(2n) \approx \frac{2}{2}T(n) = T(n)$. Se tarda $\approx$ lo mismo que antes.

Tiempos de ejecución casi lineales o mejores

Primera conclusión

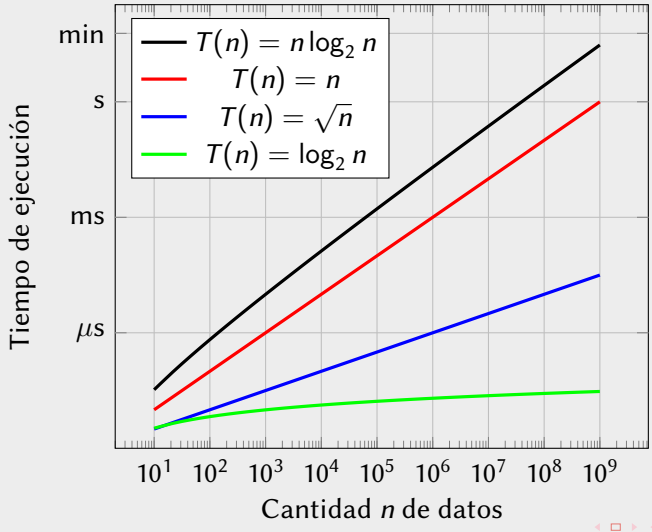
Si $T(n)$ es alguno de n, $\sqrt{n}$, $\log_2 n$ o incluso $n \log_2 n$ entonces la Ley de Wirth no aplica: los algoritmos con estos tiempos de ejecución casi lineales o mejores pueden lidiar con entradas moderadamente más grandes, aún si no aumenta la velocidad del hardware.

De forma práctica

Si cada operación tardara 1 nanosegundo (1 ns $= 10^{-9}$ s) entonces estos algoritmos podrían procesar entradas muy grandes en menos de 1 s. Qué tan grande podría ser n se muestra en esta tabla y en la siguiente gráfica.

Tiempo	$\log_2 n$	$\sqrt{n}$	n	$n \log_2 n$
1 μs	10^{301}	10^6	1000	140
1 ms	más	10^{12}	10^6	62746
1 s	más	10^{18}	10^9	39×10^6

Comparación de tiempos casi lineales o mejores



Tiempo de ejecución cuadrático

Supón que $T(n) = n^2$. Esto pasa con dos ciclos anidados como estos

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
```

Duplicar la entrada

$T(2n) = (2n)^2 = 4T(n)$. Se tarda el cuádruple que antes.

Duplicar la entrada y duplicar la velocidad

$T'(2n) = \frac{1}{2}T(2n) = \frac{4}{2}T(n) = 2T(n)$. Se tarda el doble que antes.

Tiempo de ejecución cúbico

Supón que $T(n) = n^3$. Esto pasa con tres ciclos anidados como estos

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
```

Duplicar la entrada

$T(2n) = (2n)^3 = 8T(n)$. Se tarda el óctuple que antes.

Duplicar la entrada y duplicar la velocidad

$T'(2n) = \frac{1}{2}T(2n) = \frac{8}{2}T(n) = 4T(n)$. Se tarda el cuádruple que antes.

Segunda conclusión

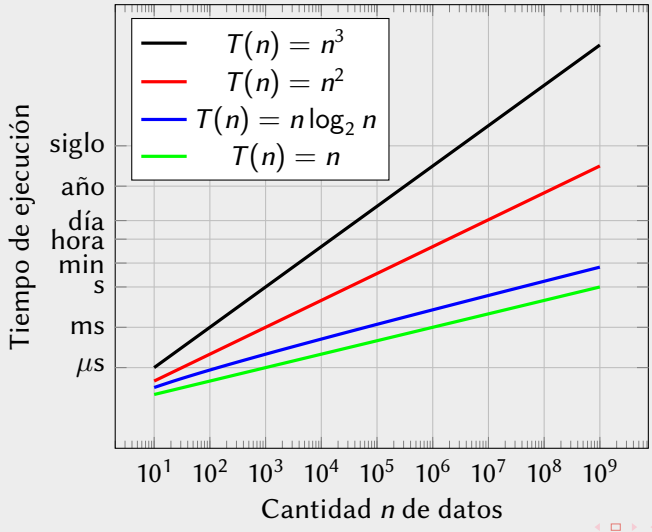
Si $T(n)$ es alguno de n^2, n^3 o algún otro polinomio, entonces la Ley de Wirth sí aplica: estos algoritmos apenas pueden lidiar con entradas moderadamente más grandes, aún si aumenta considerablemente la velocidad del hardware.

De forma práctica

Si cada operación tardara 1 ns entonces estos algoritmos podrían procesar entradas moderadamente grandes en menos de 1 hora. Qué tan grande podría ser n se muestra en esta tabla y en la siguiente gráfica.

Tiempo	n	$n \log_2 n$	n^2	n^3
1 s	10^9	39×10^6	31622	1000
1 min	6×10^{10}	12×10^8	244948	3914
1 hora	36×10^{11}	98×10^9	1897366	15326

Comparación de tiempos polinomiales



Tiempo de ejecución exponencial

Supón que $T(n) = 2^n$. Esto pasa si se quieren procesar los subconjuntos de un conjunto con n elementos o las cadenas de n bits.

Duplicar la entrada

$T(2n) = 2^{2n} = (2^n)^2 = T(n)^2$. Se tarda el cuadrado que antes.

Duplicar la entrada y duplicar la velocidad

$T'(2n) = \frac{1}{2}T(2n) = \frac{1}{2}T(n)^2$. Se tarda casi el cuadrado que antes.

Tiempo de ejecución factorial

Supón que $T(n) = n!$. Esto pasa si se quieren procesar todas las permutaciones de un conjunto con n elementos.

Duplicar la entrada

$T(2n) = (2n)! \gg (n!)^2 = T(n)^2$. MUCHO más del cuadrado que antes.

Duplicar la entrada y duplicar la velocidad

$T'(2n) = \frac{1}{2}T(2n) \gg T(n)^2$. MUCHO más del cuadrado que antes.

Tiempos de ejecución exponenciales o peores

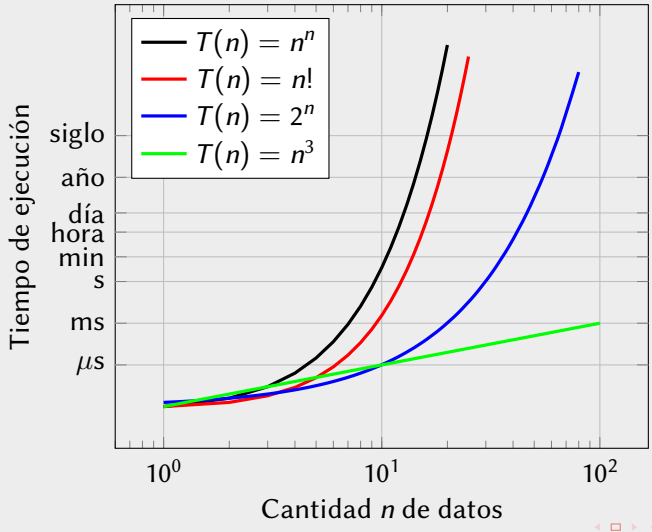
Tercera conclusión

Si $T(n)$ es alguno de 2^n, $n!$, n^n o peor, entonces la Ley de Wirth aplica con violencia: estos algoritmos no pueden lidiar con entradas remotamente grandes, aún si algún día aumentara increíblemente la velocidad del hardware.

¡Evita a toda costa usar algoritmos así!

Tiempo	2^n	$n!$	n^n
1 s	29	12	9
1 min	35	13	10
1 hora	41	15	11
1 día	46	16	12
1 año	54	18	14
1 siglo	61	20	15

Comparación de tiempos exponenciales o peores



Ejercicios: Vuelve a hacer las cuentas para...

Ejemplo (Los setentas: 1 MHz)

Época dominada por microprocesadores (como el Z80, el 6502 y el 6800) que hacían aproximadamente una operación por microsegundo (10^{-6} s).

Ejemplo (Los ochentas: 10 MHz)

Época dominada por microprocesadores (como el 8086 y el 68000) que hacían aproximadamente diez operaciones por microsegundo (10^{-7} s).

Ejemplo (Los noventas: 100 MHz)

Época dominada por microprocesadores (como el 80486 y el Pentium) que hacían aproximadamente cien operaciones por microsegundo (10^{-8} s).

Ejemplo (El futuro: 10 GHz)

Época dominada por microprocesadores que harán aproximadamente diez operaciones por nanosegundo (10^{-10} s). ¿Qué puedes concluir de esto?

Contar el número de operaciones

Inicializa un contador local (para contar las operaciones que haga una función) o global (para contar las operaciones que haga un programa).

```
int contador = 0; // si la cuenta no es muy grande
long long int contador = 0LL;  // si es muy grande
```

Después incrementa este contador cada vez que ocurra la operación.

```
contador++;
// la operacion ocurre aqui
```

El valor final del contador indicará cuántas operaciones sí ocurrieron. Con frecuencia este valor será menor al que calculemos, pues nuestros cálculos siempre los haremos considerando el peor de los casos.

Medir el tiempo de ejecución

Para esto puedes usar la biblioteca estándar de medición de tiempo.

```
#include <time.h>
```

Debes declarar una variable `clock_t tiempo` para almacenar la cuenta del tiempo y usar la función `clock()` para obtener el tiempo actual en tics.

```
tiempo = clock();          // el tiempo inicial
// todo lo que quieras medir debe ocurrir aqui
tiempo = clock() - tiempo;  // el tiempo final
```

Los tics ocurren `CLOCKS_PER_SEC` veces por segundo, así que si quieres saber cuántos segundos fueron puedes hacer una división.

```
double t = ((double) tiempo)/CLOCKS_PER_SEC;
printf("%d tics = %lf segundos\n", (int) tiempo, t);
```