Análisis de Algoritmos

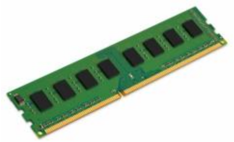
Bach. Rodolfo Mercado Gonzales
Programación Competitiva UPC

Introducción

- Las computadoras pueden ser muy rápidas, pero no infinitamente rápidas.
- La memoria puede ser barata, pero no es gratuita.
- El tiempo de ejecución y el espacio de memoria son recursos limitados.

Análisis de algoritmos

- Busca estimar los recursos que un algoritmo requiere para poder ejecutarse.
- Nos permite comparar algoritmos y saber cuál es más eficiente.
- Nos centraremos en el tiempo de ejecución y el espacio de memoria.



Tiempo de ejecución

Tiempo de ejecución

Intervalo de tiempo que le toma a un programa procesar una determinada entrada.

Tiempo de ejecución

¿ Es un buen indicador medir el tiempo de ejecución en segundos, minutos, ... ?

- Varía de acuerdo a la computadora que usemos para ejecutar el programa.
- Varía de acuerdo al tamaño de la entrada.
- Varía entre ejecución y ejecución.

El modelo RAM

- Para nuestro análisis definiremos una computadora teórica denominada Random Access Machine (RAM).
- Representa el comportamiento esencial de las computadoras.
- Las instrucciones son ejecutadas una después de otra, sin concurrencia.

El modelo RAM

Operaciones elementales

- Operaciones o instrucciones cuyo tiempo de ejecución no depende del tamaño de la entrada.
- Se realizan en tiempo unitario o constante.

  - Operaciones aritméticas.
  - Operaciones de comparación (datos primitivos).
  - Asignar el valor a una variable (datos primitivos).
  - Acceder a un elemento de un arreglo.
  - Llamada a una función y retorno de un valor.

los bucles y subprogramas poseen varias operaciones elementales.

El modelo RAM

El tiempo de ejecución $T(n)$ de un programa se mide como el total de operaciones elementales realizadas para procesar una entrada de tamaño n.

Análisis del tiempo de ejecución

Hallemos la cantidad de operaciones elementales dentro de la función.

```
// busca x en arr[]
bool search(int x, int arr[], int n) {
    for (int i = 0; i < n; ++i) {
        if (arr[i] == x) return true;
    }
    return false;
}
```

¿siempre hay n comparaciones?

Análisis del mejor caso

Menor número posible de operaciones elementales para una entrada de tamaño n.

comparación (1 op)

```
// busca x en arr[]
bool search(int x, int arr[], int n) {
    for (int i = 0; i < n; i = i + 1) {
        if (arr[i] == x) return true;
    }
    return false;
}
```

asignación (1 op)

retorno (1 op)

$$T(n) = 5$$

comparación (1 op)

acceso (1 op)

Análisis del peor caso

Máximo número posible de operaciones elementales para una entrada de tamaño n.

```
// busca x en arr[]
bool search(int x, int arr[], int n) {
    for (int i = 0; i < n; i = i + 1) {
        if (arr[i] == x) return true;
    }
    return false;
}
```

comparación (n + 1 op)

incremento (n op)

asignación (n op)

asignación (1 op)

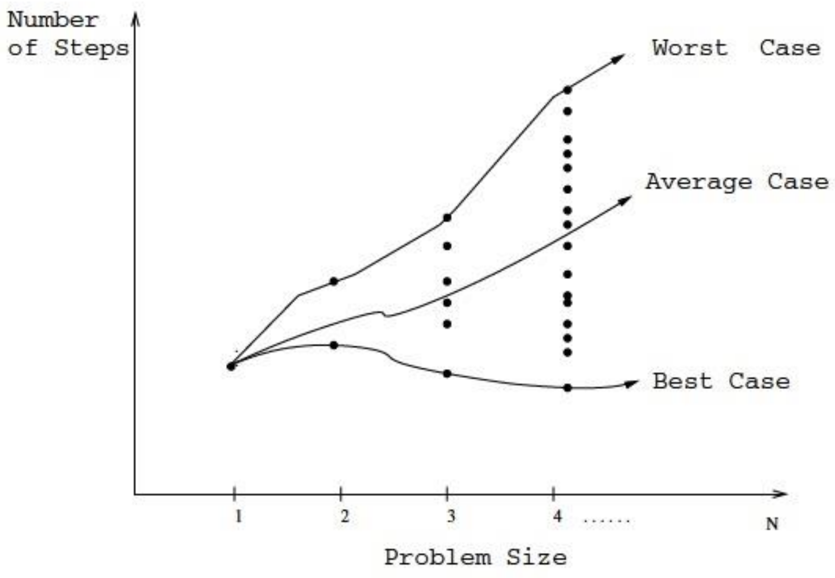
retorno (1 op)

acceso (n op)

comparación (n op)

$$T(n) = 5n + 3$$

Análisis del tiempo de ejecución



debemos analizar el peor de lo casos.

Dificultades

Calcular el número exacto de operaciones elementales requiere que el algoritmo sea especificado detalladamente.

Notación Big O

- La notación Big O es usada frecuentemente para denotar el tiempo de ejecución y la memoria usada en un algoritmo.

- Nos permite compara algoritmos sin necesidad de codificarlos.

- Nos brinda un cota superior para nuestra función $\boldsymbol{T(n)}$.

- Nos permitirá ignorar detalles que no alteran el comportamiento del tiempo de ejecución $\boldsymbol{T(n)}$.

Reglas de la notación Big O

$T(n) = 5n + 3$

- Eliminemos los términos de menor orden de nuestra función $T(n)$ $\rightarrow \ 5n$
- Eliminemos los factores constantes de nuestra función $T(n)$ $\rightarrow \ n$
- Usemos la función más pequeña posible para $g(n)$ $\rightarrow \ n \ es \ O(n) \ ya \ no \ O(n^2)$

enfoquémonos en la parte esencial de nuestro algoritmo

Notación Big O

$T(n) = 2n^2 + 100n + 6 \quad \rightarrow \quad \boldsymbol{O(n^2)}$ cuadrático

$T(n) = 5 \quad \rightarrow \quad \boldsymbol{O(1)}$ constante

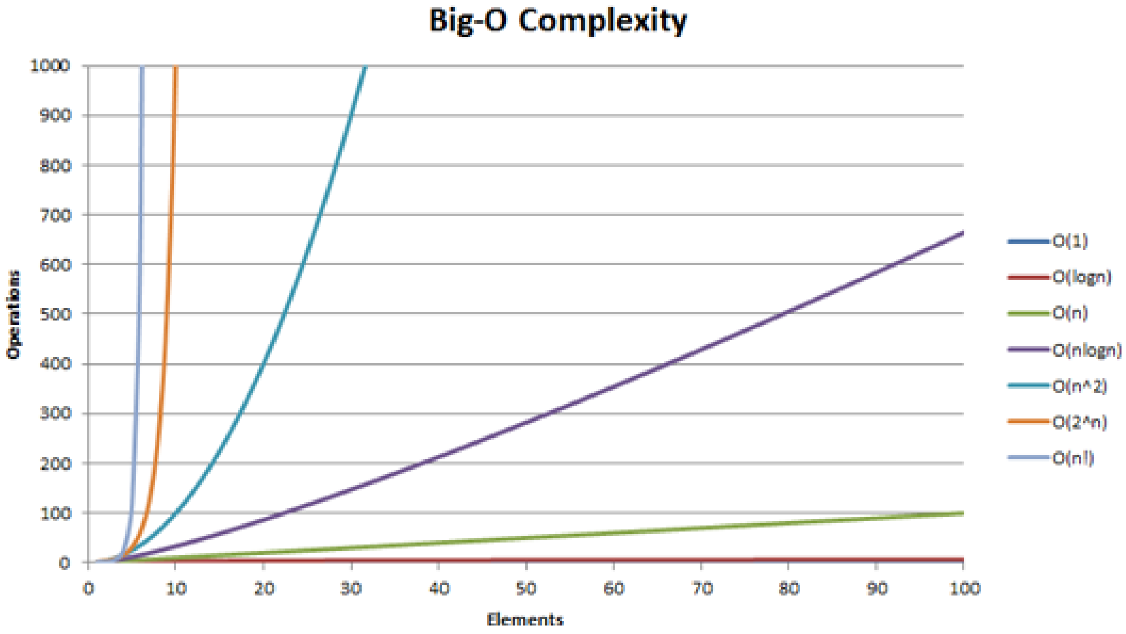
$T(n) = 3 * 2^n + 5 \quad \rightarrow \quad \boldsymbol{O(2^n)}$ exponencial

$T(n) = n + 6 \quad \rightarrow \quad \boldsymbol{O(n)}$ lineal

$T(n) = 2 \log n + 1 \quad \rightarrow \quad \boldsymbol{O(\log n)}$ logarítmico

$T(n, m) = 2 * n^2 + 3 * m \quad \rightarrow \quad \boldsymbol{O(n^2 + m)}$

Notación Big O



Consideraciones

- El tiempo de ejecución de un bucle se aproxima al número de veces (iteraciones) que el código dentro del bucle se ejecuta.

```
for (int i = 1; i <= n; i++) {
    ...
}
```

$complejidad: O(n)$

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        ...
    }
}
```

$complejidad: O(n^2)$

Consideraciones

```
for (int i = 1; i <= n+5; i++) {
    ...
}
```

$complejidad: O(n)$

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
        ...
    }
}
```

$complejidad: O(n * m)$

```
for (int i = 1; i <= 3*n; i++) {
    ...
}
```

$complejidad: O(n)$

```
for (int i = 1; i <= n; i *= 2) {
    ...
}
```

$complejidad: O(\log n)$

Consideraciones

```
for (int i = 1; i <= n; i++) {
    ...
}
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        ...
    }
}
for (int i = 1; i <= n; i++) {
    ...
}
```

$complejidad: O(n^2)$

Consideraciones

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= i; j++) {
        ...
    }
}
```

$$T(n) \sim 1 + 2 + 3 + \cdots + n = \frac{1}{2}n^2 + \frac{1}{2}n$$

$complejidad\colon\ O(n^2)$

Consideraciones

- Debemos tener cuidado al trabajar con cadenas, las operaciones de asignación, comparación y concatenación son lineales en sus tamaños.

- No todas las funciones que tenemos disponibles en C++ son $O(1)$ como aparentan.

Función	**Complejidad**
sort	$O(n \log n)$
reverse	$O(n)$
$insert$ (strings)	$O(n)$
replace (strings)	$O(n)$
$size$	$O(1)$
pop_back	$O(1)$
$back$	$O(1)$

Análisis del tiempo de ejecución

En el lenguaje **C++,** aproximadamente $\mathbf{10^8}$ operaciones elementales se ejecutan en 1 segundo.

```
clock_t ini = clock();
/*
    code
*/
clock_t fin = clock();
double run_time = (double) (fin - ini) / CLOCKS_PER_SEC;
cout << "runtime: " << fixed << setprecision(2) << run_time;
```

Análisis del tiempo de ejecución

Ejemplo 1

¿Cuántos múltiplos de 5 existen entre 1 y n ($n \leq 10^{10}$)?

Análisis del tiempo de ejecución

Solución Ingenua

Recorremos cada uno de los números del 1 a n y revisamos si es divisible por 5.

$complejidad: \; O(n)$

Análisis del tiempo de ejecución

Solución Eficiente

$$multiplos\_cinco(1, n) = \lfloor n/5 \rfloor$$

$complejidad\!: \ O(1)$

Análisis del tiempo de ejecución

Ejemplo 2

Determinar si un número n ($n \leq 10^{10}$) es primo.

Análisis del tiempo de ejecución

Solución Ingenua

Tomando como caso especial que el 1 no es primo, recorremos cada uno de los números del 2 a $n - 1$ (posibles divisores), si encontramos que alguno es divisor de n entonces el número no es primo, caso contrario será primo.

complejidad: $O(n)$

Análisis del tiempo de ejecución

Solución Eficiente

Teorema

Si n es un número compuesto, entonces n tiene al menos un divisor que es mayor que 1 y menor o igual a $\sqrt{n}$.

Análisis del tiempo de ejecución

Demostración

Sea $n = ab$; donde a, b son enteros, n un número compuesto y $1 < a \leq b < n$, entonces:

$\boldsymbol{a} \leq \sqrt{\boldsymbol{n}}$, ya que si no caeríamos en una contradicción, porque tendríamos que $a, b > \sqrt{n}$ y por ende $ab > n$.

Asimismo

$\boldsymbol{b} \geq \sqrt{\boldsymbol{n}}$, ya que si no caeríamos en una contradicción, porque tendríamos que $a, b < \sqrt{n}$ y por ende $ab < n$.

Análisis del tiempo de ejecución

Solución Eficiente

Por ende, para saber si un número $n > 1$ es primo, sólo es necesario verificar que no tenga divisores en el rango $[2, \sqrt{n}\,]$.

complejidad: $O(\sqrt{n}\,)$

Análisis del tiempo de ejecución

Ahora podemos tener una idea del orden de complejidad que requiere la solución a un problema dado una entrada de tamaño n.

Entrada	**Posible solución**
$n \leq 10$	$O(n!)$
$n \leq 20$	$O(2^n), O(2^n\ n)$
$n \leq 50$	$O(n^4)$
$n \leq 200$	$O(n^3)$
$n \leq 1000$	$O(n^2)$, $O(n^2 \log n)$
$n \leq 10^6$	$O(n), O(n \log n)$
$n \geq 10^9$	$O(1), O(\log n), O(\sqrt{n})$

límites comunes en los concursos de programación.

Ejercicios

- Codechef – Chef Jumping
- Codechef – Magic Pairs
- Codeforces – Single Push
- HackerRank – Strange Counter

Desafíos

- Codechef – Chef and Subarray
- Codechef – Count Substrings
- Hackerrank – Summing the N series
- Codeforces – Sort the Array
- Codechef – A problem onSticks

Referencias

- Thomas Cormen et al. - Introduction to Algorithms
- Steven Skiena - The Algorithm Design Manual

“The only thing worse than starting something and failing is not starting something.”

- Seth Godin