Introducción a la Computación (Matemática)

Primer Cuatrimestre de 2019

Complejidad Algorítmica

Mapa de la materia

- Programas simples en C++. ✓
- Especificación de problemas. ✓
- Correctitud de algoritmos. ✓

- Lenguaje de alto nivel: Python.
- Complejidad algorítmica: búsqueda y ordenamiento.
- Recursión algorítmica; dividir y conquistar.
- Tipos abstractos de datos: uso e implementación.
- Backtracking. (¿Heurísticas?)

Tiempo de ejecución

El tiempo de ejecución de un programa se mide en función del tamaño de la entrada.

- Ejemplo: longitud de la lista de entrada.

Notación: $T(n)$: tiempo de ejecución de un programa con una entrada de tamaño n.

- Unidad: cantidad de instrucciones.
- Ejemplo: $T(n) = c \cdot n^2$, donde c es una constante.

Tiempo de ejecución

Podemos considerar tres casos del tiempo de ejecución:

- peor caso: tiempo máximo de ejecución para alguna entrada;
- mejor caso: tiempo mínimo de ejecución para alguna entrada;
- caso promedio: tiempo de ejecución para la *entrada promedio*.

Vamos a ver sólo el peor caso: $T(n)$ es una cota superior del tiempo de ejecución para entradas arbitrarias de tamaño n.

Cálculo del tiempo de ejecución

Instrucciones minimales: lectura/escritura de una variable o de una posición en un arreglo, consulta de la longitud de un arreglo, operaciones simples de tipos básicos. $T_S(n) = 1$

- Ej: $T_{x \leftarrow 2*y+1}(n) = T_y + T_* + T_+ + T_\leftarrow = 1+1+1+1 = 4$

Secuencialización: $T_{S_1;S_2}(n) = T_{S_1}(n) + T_{S_2}(n)$

- Ej: $T_{x \leftarrow y+1;\, y \leftarrow 0}(n) = T_{x \leftarrow y+1}(n) + T_{y \leftarrow 0}(n) = 3 + 1 = 4$

Condicional: $T_{\text{if } (B)\ S_1 \text{ else } S_2}(n) = T_B(n) + \text{máx}(T_{S_1}(n), T_{S_2}(n))$

Ciclo: $T_{\text{while } (B)\ S}(n) = T_{B_0}(n) + \sum_{\eta \in \text{iteraciones}} \left(T_{S_\eta}(n) + T_{B_\eta}(n) \right)$

Ejemplo: Problema de Búsqueda

Encabezado: $Buscar : x \in \mathbb{Z} \times A \in \mathbb{Z}[\,] \to está \in \mathbb{B} \times pos \in \mathbb{Z}$

Precondición: $\{A = A_0 \wedge x = x_0\}$

Poscondición: $\{(está = true \wedge 0 \leq pos < |A_0| \wedge A_0[pos] = x_0) \vee$
$(está = false \wedge (\forall i)(0 \leq i < |A_0| \Rightarrow A_0[i] \neq x_0))\}$

$está \leftarrow false$ (1)
$pos \leftarrow -1$ (1)
$i \leftarrow 0$ (1)
while $(i < |A|)$ { (3)
  if $(A[i] = x)$ { (4)
    $está \leftarrow$ true (1)
    $pos \leftarrow i$ (2)
  }
  $i \leftarrow i + 1$ (3)
} (while: $i = 0, 1, \ldots, |A| - 1 \rightsquigarrow |A|$ iteraciones)

$$T(|A|) = 1 + 1 + 1 + 3 + \Sigma_{0 \leq i < |A|}\big(4 + 1 + 2 + 3 + 3\big)$$
$$= 6 + \Sigma_{0 \leq i < |A|} 13 = 6 + 13\,|A|$$

Orden del tiempo de ejecución

Definición: Decimos que $T(n) \in O(f(n))$ si existen constantes enteras positivas c y n_0 tales que para $n \geq n_0$, $T(n) \leq c \cdot f(n)$.

Ejemplo: $T(n) = 3n^3 + 2n^2$.

$T(n) \in O(n^3)$, dado que si tomamos $n_0 = 1$ y $c = 5$, vale que para $n \geq 1,\ T(n) \leq 5 \cdot n^3$.

Ejemplo: Problema de búsqueda
$T(|A|) = 6 + 13\,|A|\ \in O(|A|)$ (orden lineal)

Por eso se lo conoce como algoritmo de búsqueda lineal.

Complejidad temporal

Propiedades de O:

- Regla de la suma:
  Si $f_1 \in O(g)$ y $f_2 \in O(h) \Rightarrow f_1 + f_2 \in O(\text{máx}(g, h))$.
  - Ej: $f_1 \in O(n^2)$ y $f_2 \in O(n)$, luego $f_1 + f_2 \in O(n^2)$.
  - Ej: $f_1 \in O(1)$ y $f_2 \in O(1)$, luego $f_1 + f_2 \in O(1)$.

- Regla del producto:
  Si $f_1 \in O(g)$ y $f_2 \in O(h) \Rightarrow f_1 \cdot f_2 \in O(g \cdot h)$.
  - Ej: $f_1 \in O(n^2)$ y $f_2 \in O(n)$, luego $f_1 \cdot f_2 \in O(n^3)$.
  - Ej: $f_1 \in O(n)$ y $f_2 \in O(1)$, luego $f_1 \cdot f_2 \in O(n)$.

Cálculo de órdenes de complejidad

Instrucciones minimales: lectura/escritura de una variable o de una posición en un arreglo, longitud de un arreglo, operaciones simples de tipos básicos. Orden constante: $O(1)$.

Secuencialización: Si S_1 y S_2 tienen $O(f)$ y $O(g)$, resp., entonces $S_1; S_2$ tiene $O(f) + O(g) = O(\max(f, g))$.

Condicional: Si B, S_1 y S_2 tienen $O(f)$, $O(g)$ y $O(h)$, if (B) S_1 else S_2 tiene $O(f) + O(\max(g, h)) = O(\max(f, g, h))$.

Ciclo: Si B y S tienen $O(f)$ y $O(g)$, y se ejecutan $O(h)$ veces, entonces while (B) S tiene $O(f + g) \cdot O(h) = O((f + g) \cdot h)$.

Problema de búsqueda

Encabezado: $Buscar : x \in \mathbb{Z} \times A \in \mathbb{Z}[\,] \to está \in \mathbb{B} \times pos \in \mathbb{Z}$

Precondición: $\{A = A_0 \wedge x = x_0\}$

Poscondición: $\{(está = true \wedge 0 \leq pos < |A_0| \wedge A_0[pos] = x_0) \vee$
$(está = false \wedge (\forall i)(0 \leq i < |A_0| \Rightarrow A_0[i] \neq x_0))\}$

$está \leftarrow false$ $\quad O(1)$
$pos \leftarrow -1$ $\quad O(1)$
$j \leftarrow 0$ $\quad O(1)$
while $(j < |A|)$ { $\quad O(1)$
    if $(A[j] = x)$ { $\quad O(1)$
        $está \leftarrow$ true $\quad O(1)$
        $pos \leftarrow j$ $\quad O(1)$
    }
    $j \leftarrow j + 1$ $\quad O(1)$
} while: $O(|A|)$ iteraciones

¿Cuál es el orden de complejidad?

$T(|A|) \in O(|A|)$ **Búsqueda lineal**

¿Y si agregamos "$\wedge \neg está$" a la guarda del while?

En este algoritmo, cortar antes la ejecución puede ahorrar tiempo, pero no cambia el **orden en el peor caso**.

¿Cuán eficientes son estos algoritmos si A está ordenado?

Veamos un algoritmo de búsqueda para listas ordenadas.

4	7	23	41	44	59	97	134	165	187	210	212	249	280	314
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Buscamos el número **97**...

4	7	23	41	44	59	97	134
0	1	2	3	4	5	6	7

165	187	210	212	249	280	314
8	9	10	11	12	13	14

4	7	23	41
0	1	2	3

44	59	97	134
4	5	6	7

44	59
4	5

97	134
6	7

97
6

134
7

97
6

$\checkmark$

Búsqueda binaria

¿Cuál es el comportamiento detrás de este algoritmo?

Si la lista está ordenada, entonces en cada paso puedo partir la lista en:

a) la mitad que puede contener el elemento; y
b) la mitad que no puede contenerlo.

Indefectiblemente, se llega a un punto en que la lista ya no puede ser dividida (tiene un solo elemento) y, o bien el elemento es el buscado o no.

Encabezado: $Buscar : x \in \mathbb{Z} \times A \in \mathbb{Z}[\,] \to está \in \mathbb{B} \times pos \in \mathbb{Z}$

Precondición: $\{A = A_0 \wedge x = x_0 \wedge$
$(\forall i)(0 \leq i < |A| - 1 \Rightarrow A[i] \leq A[i+1])\}$

Poscondición: $\{(está = true \wedge 0 \leq pos < |A_0| \wedge A_0[pos] = x_0) \vee$
$(está = false \wedge (\forall i)(0 \leq i < |A_0| \Rightarrow A_0[i] \neq x_0))\}$

$(está, pos) \leftarrow (false, -1)$
$(izq, der) \leftarrow (0, |A| - 1)$
while $(izq < der)$ {
  $med \leftarrow (izq + der)$ div 2
  if $(A[med] < x)$ {
    $izq \leftarrow med + 1$
  } else {
    $der \leftarrow med$
  }
}
if $(x = A[izq])$ {
  $(está, pos) \leftarrow (true, izq)$
}

¿Cuál es el orden de complejidad?

$T(|A|) \in O(\log |A|)$
orden logarítmico

Búsqueda binaria

Para ver que el orden es logarítmico, basta observar que la función variante $fv = der - izq$ decrece aproximadamente a la mitad en cada iteración:

Sea $fv = der - izq$ al comienzo de una iteración.
Al final de la misma, pueden ocurrir dos cosas:

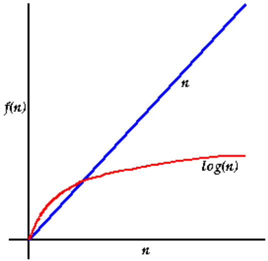
- $fv' = der - \lfloor \frac{izq+der}{2} \rfloor - 1 \approx \lfloor \frac{fv}{2} \rfloor$
- $fv' = \lfloor \frac{izq+der}{2} \rfloor - izq \approx \lfloor \frac{fv}{2} \rfloor$

En ambos casos, fv termina valiendo aproximadamente la mitad que al principio de la iteración.

Como el ciclo termina cuando $fv \leq 0$, el cuerpo del ciclo se ejecuta $O(\log_2 |A|)$ veces. $\square$

Obs.: La base del log es irrelevante para el orden de T.

Búsqueda lineal vs. binaria



¿Cuán importante es la diferencia entre $O(\log n)$ y $O(n)$?

Depende de nuestro contexto...

- ¿Cuál es el tamaño del listado en el cual haremos la búsqueda? (FCEyN vs. ANSES)
- ¿Cuánto cuesta cada consulta individual? (Lectura en memoria vs. consulta por Internet)
- ¿Cuántas veces vamos a necesitar hacer esta búsqueda? (una vez por mes vs. millones de veces por día)

¿Cuál programa usamos?

Objetivos contrapuestos

Para resolver un problema, queremos un programa...

1. que sea fácil de programar (que escribirlo nos demande poco tiempo, que sea simple y fácil de entender);
2. que consuma pocos recursos: tiempo y espacio (memoria, disco rígido).

En general priorizamos un objetivo sobre el otro:

- para programas que correrán pocas veces, priorizamos el objetivo 1;
- para programas que correrán muchas veces, priorizamos el objetivo 2.

Repaso de la clase de hoy

- Tiempo de ejecución, medido en cantidad de operaciones.
- Peor caso, mejor caso y caso promedio.
- Cálculo del tiempo de ejecución.
- Orden del tiempo de ejecución.
- Algoritmos de búsqueda.

Próximos temas

- Algoritmos de ordenamiento.