Introducción

IIC2283

El objetivo de este curso

Introducir técnicas tanto para el diseño como para el análisis de la complejidad computacional de un algoritmo

- Técnicas básicas y avanzadas

Se dará énfasis a:

- La compresión del modelo computacional sobre el cual se diseña y analiza un algoritmo
- El uso de estructuras de datos adecuadas para la implementación de un algoritmo
- El uso de ejemplos de distintas áreas para mostrar las potencialidades de las técnicas estudiadas

Algunas consideraciones importantes

Al diseñar un algoritmo debemos considerar el modelo de computación sobre el cual será implementado.

- ¿Qué operaciones podemos realizar en el modelo?

Al analizar la complejidad computacional de un algoritmo también debemos considerar el modelo de computación.

- ¿Qué operaciones consideramos al analizar la complejidad de un algoritmo?

Algunas consideraciones importantes

En general, el análisis de la complejidad de un algoritmo se realiza considerando un tipo particular de entradas.

- El peor caso es muy utilizado, pero también podemos considerar el caso promedio

Al estudiar un problema debemos tener en cuenta que cotas inferiores se puede demostrar para su complejidad

- Estas cotas inferiores dependen del modelo de computación considerado

Algunas consideraciones importantes

Debemos considerar modelos de computación que representen el funcionamiento de una arquitectura de computadores.

- Por ejemplo, debemos consider acceso directo a los datos y la diferencia de costo entre el uso de memoria principal y secundaria

Vamos a considerar dos ejemplos que nos servirán para ilustrar los puntos anteriores: ordenación y la multiplicación de números enteros

Ordenación de una lista

El siguiente es una algoritmo clásico para ordenar una lista L de números enteros (de menor a mayor).

InsertionSort($L[1 \ldots n]$: lista de números enteros)
    for $i := 2$ **to** n **do**
        $j := i - 1$
        while $j \geq 1$ **and** $L[j] > L[j+1]$ **do**
            $aux := L[j]$
            $L[j] := L[j+1]$
            $L[j+1] := aux$
            $j := j - 1$
    return L

Análisis de la complejidad de insertion sort

Consideramos la comparación como la operación a contar, la cual tiene costo 1.

Dada una lista L con n números enteros.

- ¿Cuál es el peor caso del algoritmo?
- ¿Cuantas comparaciones realiza el algoritmo en el peor caso, como una función de n?

¿Qué otras operaciones son realizadas por el algoritmo? ¿Cambia el tiempo de ejecución del algoritmo si las consideramos?

Una versión mejorada de insertion sort

Suponiendo que $L[1 \cdots (i-1)]$ ya ha sido ordenada (de menor a mayor), el paso básico del algoritmo es encontrar la posición donde debería ser ubicado $L[i]$

- Además el algoritmo debe colocar $L[i]$ en esta posición

Para disminuir el tiempo de ejecución del algoritmo podríamos utilizar búsqueda binaria para encontrar la posición correcta para $L[i]$

Una versión mejorada de insertion sort

Consideramos nuevamente la comparación como la operación a contar.

Dado que búsqueda binaria realiza $O(\log_2(i))$ comparaciones para encontrar la posición correcta para $L[i]$, el algoritmo mejorado es de orden $O(n \cdot \log_2(n))$

- Esto puede ser deducido utilizando lo siguiente:

$$\begin{aligned}
\sum_{i=1}^{n} \log_2(i) &= \log_2\left(\prod_{i=1}^{n} i\right) \\
&= \log_2(n!) \\
&\leq \log_2(n^n) \\
&= n \cdot \log_2(n)
\end{aligned}$$

¿Una versión mejorada de insertion sort?

¿Es más eficiente el algoritmo que utiliza búsqueda binaria?

- ¿Ve algún problema en este algoritmo?

Un problema con este algoritmo: ¿Cómo colocar de manera eficiente $L[i]$ en la posición correcta en $L[1 \ldots (i-1)]$?

- Un algoritmo ingenuo toma tiempo lineal en este paso, por lo que el tiempo total del algoritmo sería $O(n^2)$, el mismo orden que para insertion sort

¿Una versión mejorada de insertion sort?

Dos lecciones importantes

- Una operación puede ser cambiada por otra en un algoritmo, esto debe tenerse en cuenta al momento de analizar su complejidad.
- El uso de estructuras de datos es fundamental para implementar de manera eficiente las operaciones requeridas por un algoritmo.

Suma de números enteros

Sean a y b dos números enteros con $n \geq 1$ dígitos cada uno.

Primero queremos obtener $c = a + b$

Vamos a utilizar el algoritmo usual para calcular c

- Consideramos la suma de dos dígitos, la comparación de dos dígitos y la resta de un número con a lo más dos dígitos con uno de un dígito como las operaciones a contar, cada una con costo 1. ¿Tiene sentido suponer que todas tienen el mismo costo?
- ¿Cuál es el peor caso para el algoritmo usual? ¿Cuántas operaciones realiza el algoritmo en este caso?
- ¿Cuántos dígitos puede tener c?

Multiplicación de números enteros

Ahora queremos obtener $d = a \cdot b$

Primero vamos a utilizar el algoritmo usual para calcular d

- Consideramos la suma y la multiplicación de dígitos como las operaciones a contar, ambas con costo 1. ¿Tienes sentido suponer que ambas operaciones tienen el mismo costo?
- ¿Cuál es el peor caso para el algoritmo usual? ¿Cuántas operaciones realiza el algoritmo en este caso?
- ¿Cuántos dígitos puede tener d?

El algoritmo de multiplicación de Karatsuba

Suponga que a y b son dos números con n dígitos cada uno, donde n es una potencia de 2.

Podemos representar a y b de la siguiente forma:

$$\begin{aligned} a &= a_1 \cdot 10^{\frac{n}{2}} + a_2 \\ b &= b_1 \cdot 10^{\frac{n}{2}} + b_2 \end{aligned}$$

Tenemos entonces que:

$$a \cdot b = a_1 \cdot b_1 \cdot 10^n + (a_1 \cdot b_2 + a_2 \cdot b_1) \cdot 10^{\frac{n}{2}} + a_2 \cdot b_2$$

El algoritmo de multiplicación de Karatsuba

Para calcular $a \cdot b$ entonces debemos calcular las siguientes multiplicaciones:

1. $a_1 \cdot b_1$
2. $a_1 \cdot b_2$
3. $a_2 \cdot b_1$
4. $a_2 \cdot b_2$

Obtenemos entonces un algoritmo recursivo

- Para resolver el caso de largo n realizamos 4 llamadas para los casos de largo $\frac{n}{2}$

¿Cómo llamamos a este tipo de algoritmos?

- ¿Cómo medimos su complejidad?
- ¿Con las 4 llamadas anteriores obtenemos un algoritmo más eficiente que el algoritmo usual de multiplicación?

La idea clave en el algoritmo de Karatsuba

Para calcular $a \cdot b$ realizamos las siguientes multiplicaciones:

1. $c_1 = a_1 \cdot b_1$
2. $c_2 = a_2 \cdot b_2$
3. $c_3 = (a_1 + a_2) \cdot (b_1 + b_2)$

Tenemos entonces que:

$$a \cdot b \quad = \quad c_1 \cdot 10^n + (c_3 - (c_1 + c_2)) \cdot 10^{\frac{n}{2}} + c_2$$

¿Cuántas operaciones realiza este algoritmo?

El tiempo de ejecución del algoritmo de Karatsuba

$T(n)$: número de operaciones realizadas por el algoritmo de Karatsuba para dos números de entrada con n dígitos cada uno en el peor caso

Para determinar el orden de $T(n)$ utilizamos una ecuación de recurrencia:

$$T(n) = \begin{cases} 1 & n = 1 \\ 3 \cdot T(\frac{n}{2}) + e \cdot n & n > 1 \end{cases}$$

El tiempo de ejecución del algoritmo de Karatsuba

¿Qué supuestos realizamos al formular esta ecuación?

- n es una potencia de 2
- $(a_1 + a_2)$ y $(b_1 + b_2)$ tienes $\frac{n}{2}$ dígitos cada uno

¿Qué representa la constante e? Utilizamos e para codificar el números de operaciones necesarias para:

- Calcular $(a_1 + a_2)$, $(b_1 + b_2)$, $(c_1 + c_2)$ y $(c_3 - (c_1 + c_2))$
- Construir $a \cdot b$ a partir de c_1, c_2 y $(c_3 - (c_1 + c_2))$, lo cual puede tomar tiempo lineal en el peor caso. ¿Por qué?

Resolviendo una ecuación de recurrencia

¿Cómo resolvemos la ecuación de recurrencia para $T(n)$

Como $n = 2^k$ tenemos que:

$$T(2^k) = \begin{cases} 1 & k = 0 \\ 3 \cdot T(2^{k-1}) + e \cdot 2^k & k > 0 \end{cases}$$

Resolviendo una ecuación de recurrencia

Para resolver la ecuación simplemente la desarrollamos:

$$
\begin{aligned}
T(2^k) &= 3 \cdot T(2^{k-1}) + e \cdot 2^k \\
&= 3 \cdot (3 \cdot T(2^{k-2}) + e \cdot 2^{k-1}) + e \cdot 2^k \\
&= 3^2 \cdot T(2^{k-2}) + 3 \cdot e \cdot 2^{k-1} + e \cdot 2^k \\
&= 3^2 \cdot (3 \cdot T(2^{k-3}) + e \cdot 2^{k-2}) + 3 \cdot e \cdot 2^{k-1} + e \cdot 2^k \\
&= 3^3 \cdot T(2^{k-3}) + 3^2 \cdot e \cdot 2^{k-2} + 3 \cdot e \cdot 2^{k-1} + e \cdot 2^k \\
&= 3^3 \cdot T(2^{k-3}) + e \cdot 2^k \cdot \left(\left(\frac{3}{2} \right)^2 + \frac{3}{2} + 1 \right) \\
&\dots \\
&= 3^i \cdot T(2^{k-i}) + e \cdot 2^k \cdot \sum_{j=0}^{i-1} \left(\frac{3}{2} \right)^j
\end{aligned}
$$

Resolviendo una ecuación de recurrencia

Considerando $i = k$ obtenemos:

$$T(2^k) = 3^k \cdot T(1) + e \cdot 2^k \cdot \sum_{j=0}^{k-1} \left(\frac{3}{2}\right)^j$$

Tenemos entonces que:

$$\begin{aligned} T(2^k) &= 3^k + e \cdot 2^k \cdot \left(\frac{(\frac{3}{2})^k - 1}{\frac{3}{2} - 1}\right) \\ &= 3^k + 2 \cdot e \cdot 2^k \cdot \left(\frac{3^k - 2^k}{2^k}\right) \\ &= 3^k + 2 \cdot e \cdot (3^k - 2^k) \\ &= 3^k \cdot (1 + 2 \cdot e) - 2 \cdot e \cdot 2^k \end{aligned}$$

Considerando que $k = \log_2(n)$ obtenemos:

$$T(n) = 3^{\log_2(n)} \cdot (1 + 2 \cdot e) - 2 \cdot e \cdot n$$

Resolviendo una ecuación de recurrencia

Finalmente tenemos que:

$$\begin{aligned} 3^{\log_2(n)} &= 2^{\log_2(3^{\log_2(n)})} \\ &= 2^{\log_2(n)\cdot\log_2(3)} \\ &= (2^{\log_2(n)})^{\log_2(3)} \\ &= n^{\log_2(3)} \end{aligned}$$

De lo cual concluimos que $T(n) = n^{\log_2(3)} \cdot (1 + 2 \cdot e) - 2 \cdot e \cdot n$

- Vale decir, $T(n)$ es $O(n^{\log_2(3)})$

¿Es válido este resultado para todo n? ¿Qué sucede si eliminamos los supuestos?

- Vamos a estudiar algunas herramientas para contestar este tipo de preguntas