Introducción a la algoritmia I

José de Jesús Lavalle Martínez

Benemérita Universidad Autónoma de Puebla
Facultad de Ciencias de la Computación
Maestría en Ciencias de la Computación
Análisis y Diseño de Algoritmos
MCOM 20300

Otoño 2020

Contenido

Variaciones entre ejemplares del mismo tamaño I

El tiempo que consume un algoritmo o el espacio de almacenamiento que usa, puede variar considerablemente entre dos ejemplares distintos del mismo tamaño. Para ilustrar esto, considere dos algoritmos de ordenamiento (ascendente) elementales: ordenamiento por **inserción** y ordenamiento por **selección**.

Variaciones entre ejemplares del mismo tamaño II

procedure INSERT($T[1 \ldots n]$)

```
1  for i ← 2 to n
2  do x ← T[i]
3     j ← i − 1
4     while j > 0 and x < T[j]
5     do T[j + 1] ← T[j]
6        j ← j − 1
7     T[j + 1] ← x
```

procedure SELECT($T[1 \ldots n]$)

```
1  for i ← 1 to n − 1
2  do minj ← i
3     minx ← T[i]
4     for j ← i + 1 to n
5     do if T[j] < minx
6           then minj ← j
7                minx ← T[j]
8     T[minj] ← T[i]
9     T[i] ← minx
```

Variaciones entre ejemplares del mismo tamaño III

Ejercicio 1

1. Simule la operación de los algoritmos de ordenamiento sobre algunos arreglos pequeños, para asegurarse que entiende cómo trabajan.
2. Simule los algoritmos de ordenamiento por inserción y por selección sobre los siguientes dos arreglos: $U = [1, 2, 3, 4, 5, 6]$ y $V = [6, 5, 4, 3, 2, 1]$ ¿Sobre cuál de los arreglos U o V se ejecuta más rápido ordenamiento por inserción?. La misma pregunta pero ahora considerando ordenamiento por selección. Justifique sus respuestas.
3. Suponga que trata de "ordenar" el arreglo $W = [1, 1, 1, 1, 1, 1]$ cuyos elementos son iguales, usando: (a) ordenamiento por inserción y (b) ordenamiento por selección. ¿Cómo se compara esto a ordenar los arreglos U y V del ejercicio 1.2?

Variaciones entre ejemplares del mismo tamaño IV

- El ciclo principal en ordenaminto por inserción, busca sucesivamente cada elemento del arreglo desde el segundo hasta el $n-$ésimo y lo inserta apropiadamente entre sus predecesores en el arreglo.

Variaciones entre ejemplares del mismo tamaño IV

- Ordenamiento por selección trabaja tomando al elemento más pequeño en el arreglo y llevándolo al inicio; luego toma al siguiente más pequeño y lo pone en la segunda posición en el arreglo y así sucesivamente.

Variaciones entre ejemplares del mismo tamaño V

- Sean U y V dos arreglos de n elementos, tales que U está ordenado ascendentemente y V está ordenado descendentemente. Si resolvió bien el ejercicio 1.2 habrá notado que ambos algoritmos consumen más tiempo sobre V que sobre U.

Variaciones entre ejemplares del mismo tamaño V

- En efecto, el arreglo V representa el peor caso posible para estos dos algoritmos: ningún arreglo de n elementos requiere más trabajo.

Variaciones entre ejemplares del mismo tamaño V

- Sin embargo, el tiempo requerido por el algoritmo de ordenamiento por selección no es muy sensible al orden original del arreglo que ha de ordenarse: la prueba "**if** $T[j] < minx$" se ejecuta exactamente el mismo número de veces en cada caso.

Variaciones entre ejemplares del mismo tamaño VI

- La variación en tiempo de ejecución, es debida solamente al número de veces que son ejecutadas las asignaciones en la parte **then** de dicha prueba.

Variaciones entre ejemplares del mismo tamaño VI

- Si se programa este algoritmo y se ejecuta sobre una máquina, encontrará que el tiempo requerido para ordenar un cierto número de elementos no variará más del 15 % cualquiera que sea el orden inicial de los elmentos a ordenar.

Variaciones entre ejemplares del mismo tamaño VI

- Como se mostrará posteriormente el tiempo requerido por $select(T)$ es cuádratico, sin importar el orden inicial de los elementos.

Variaciones entre ejemplares del mismo tamaño VII

- La situación es diferente si comparamos los tiempos que toma el algoritmo de ordenamiento por inserción sobre los dos arreglos.

Variaciones entre ejemplares del mismo tamaño VII

- Ya que la condición que controla el ciclo **while** es falsa siempre desde el principio, $insert(U)$ es muy rápido y consume tiempo lineal.

Variaciones entre ejemplares del mismo tamaño VII

- Por otro lado, $insert(V)$ consume tiempo cudrático porque el ciclo **while** se ejecuta $i - 1$ veces para cada valor de i.

Variaciones entre ejemplares del mismo tamaño VII

- La variación en tiempo entre estos dos ejemplares es por lo tanto considerable. Más aún, esta variación aumenta conforme aumenta el número de elementos a ordenar.

Variaciones entre ejemplares del mismo tamaño VIII

- En alguna implementación del algoritmo de ordenamiento por inserción, se encontró que consume menos de un quinto de segundo para ordenar un arreglo de 5000 elementos que originalmente estaban en orden ascendente.

Variaciones entre ejemplares del mismo tamaño VIII

- Pero consumió tres y medio minutos en ordenar un arreglo con la misma cantidad de elementos, pero inicialmente en orden descendente, es decir, mil veces más.

Variaciones entre ejemplares del mismo tamaño IX

- Si pueden ocurrir variaciones tan grandes, ¿Cómo podemos hablar del tiempo que consume un algoritmo solamente en términos del tamaño del ejemplar a resolver?

Variaciones entre ejemplares del mismo tamaño IX

- La respuesta es que usualmente consideraremos el **peor caso** del algoritmo, esto es, para cada tamaño de ejemplar sólo consideraremos aquellos ejemplares sobre los que el algoritmo requiere más tiempo.

Variaciones entre ejemplares del mismo tamaño IX

- Esto es por lo que dijimos anteriormente que un algoritmo debe ser capaz de resolver todo ejemplar de tamaño n en no más de $ct(n)$ segundos, para una constante apropiada c que depende de la implementación.

Variaciones entre ejemplares del mismo tamaño IX

- Si decimos que un algoritmo se ejecuta en un tiempo en el orden de $t(n)$, implicitamente tenemos en mente el peor caso.

Variaciones entre ejemplares del mismo tamaño X

- El análisis del peor caso es apropiado para un algoritmo cuyo tiempo de respuesta es crítico.

Variaciones entre ejemplares del mismo tamaño X

- Por ejemplo, si se trata de controlar una planta de energía nuclear, es crucial conocer un límite superior sobre el tiempo de respuesta del sistema, no importando el ejemplar particular que se resolverá.

Operación elemental I

- Una **operación elemental** es aquella cuyo tiempo de ejecución se puede acotar superiormente por una constante, que sólo depende de la implementación que en particular use una máquina, un lenguaje de programación y así sucesivamente.

Operación elemental I

- Así la constante no depende ni del tamaño, ni de los otros parámetros del ejemplar que se está considerando.

Operación elemental I

- Dado que nos interesa el tiempo de ejecución de algoritmos que están por debajo de una constante multiplicativa, lo único que importa en el análisis es el número de operaciones elementales ejecutadas, no el tiempo exacto que requiere cada uno de ellos.

Operación elemental II

- Por ejemplo, suponga que cuando analizamos un algoritmo, encontramos que para resolver un ejemplar de un cierto tamaño se requieren ejecutar a adiciones, m multiplicaciones y s instrucciones de asignamiento.

Operación elemental II

- Por ejemplo, suponga que cuando analizamos un algoritmo, encontramos que para resolver un ejemplar de un cierto tamaño se requieren ejecutar a adiciones, m multiplicaciones y s instrucciones de asignamiento.
- Suponga que también sabemos que una adición nunca consume más de t_a microsegundos, una multiplicación nunca más de t_m microsegundos y un asignamiento nunca más de t_s microsegundos, donde t_a, t_m y t_s son constantes que dependen de la máquina usada.

Operación elemental II

- Por ejemplo, suponga que cuando analizamos un algoritmo, encontramos que para resolver un ejemplar de un cierto tamaño se requieren ejecutar a adiciones, m multiplicaciones y s instrucciones de asignamiento.
- Suponga que también sabemos que una adición nunca consume más de t_a microsegundos, una multiplicación nunca más de t_m microsegundos y un asignamiento nunca más de t_s microsegundos, donde t_a, t_m y t_s son constantes que dependen de la máquina usada.
- Por tanto, adición, multiplicación y asignamiento pueden ser consideradas operaciones elementales.

Operación elemental III

El tiempo total t que requiere nuestro algoritmo puede acotarse mediante

$$\begin{aligned} t &\leq at_a + mt_m + st_s \\ &\leq \max(t_a, t_m, t_s) \times (a + m + s), \end{aligned}$$

esto es, t está acotado por un múltiplo constante del número de operaciones elementales que son ejecutadas.

Costo unitario

Ya que no es importante el tiempo requerido por cada operación elemental, simplificaremos diciendo que las operaciones elementales se pueden ejecutar con **costo unitario**.

Advertencias I

- En la descripción de un algoritmo, una sola línea de código puede corresponder a un número variable de operaciones elementales.

Advertencias I

- Por ejemplo, si T es un arreglo de n elementos $(n > 0)$, el tiempo requerido para calcular

$$x \leftarrow \min\{T[i] | 1 \leq i \leq n\}$$

incrementa con n ya que es una abreviación para

function $\textsc{Min}(T[1 \ldots n])$
1 $x \leftarrow T[1]$
2 **for** $i \leftarrow 2$ **to** n
3 **do if** $T[i] < x$
4 **then** $x \leftarrow T[i]$
5 **return** x

Advertencias II

- Algunas operaciones matemáticas son tan complejas que no se pueden considerar elementales.

Advertencias II

- Si nos permitiéramos contar con costo unitario las operaciones calcular un factorial y prueba por divisibilidad, sin considerar el tamaño de los operandos,

Advertencias II

- entonces por el teorema de Wilson (el cual enuncia que el entero n divide a $(n-1)!+1$ si y sólo si n es primo para todo $n > 1$) podríamos probar la primalidad de un entero con eficiencia sorprendente.

```
function WILSON(n)
1   if n divide exactamente a  (n − 1)! + 1
2      then return true
3      else  return false
```

Advertencias III

- El ejemplo al inicio de esta subsección sugiere que podemos considerar como de costo unitario a las operaciones adición y multiplicación, ya que se asume que el tiempo requerido por estas operaciones se puede acotar mediante una constante.

Advertencias III

- No obstante, estas operaciones no son elementales ya que el tiempo necesario para ejecutarlas incrementa con el tamaño de los operandos.

Advertencias III

- En la práctica, por otro lado, pudiera ser adecuado considerarlas como operaciones elementales, siempre y cuando los operandos involucrados sean de un tamaño razonable en los ejemplares que esperamos encontrar.

Advertencias IV

- Dos ejemplos ilustrarán lo que queremos decir.

function $\textsc{Sum}(n)$

```
1  sum ← 0
2  for i ← 1 to n
3  do sum ← sum + i
4  return sum
```

function $\textsc{FibIter}(n)$

```
1  i ← 1
2  j ← 0
3  for k ← 1 to n
4  do j ← i + j
5     i ← j − i
6  return j
```

Advertencias IV

- En el algoritmo llamado SUM el valor de *sum* permanece razonable para todos los ejemplares que en la práctica se esperan. Si estamos usando una máquina con palabras de 32 bits, todas las adiciones pueden ser ejecutadas directamente siempre y cuando n no sea mayor a 65535.

Advertencias V

- En teoría, el algoritmo debe trabajar para *todos* los valores posibles de n. Pero, ninguna máquina real puede ejecutar estas adiciones con costo unitario si n es elegido suficientemente grande.

Advertencias V

- El análisis del algoritmo por tanto debe depender del dominio previsto de la aplicación (no del dominio del problema).

Advertencias VI

- La situación es diferente en el caso de FibIter. Aquí es suficiente tomar $n = 47$ para que la última adición "$j \leftarrow i + j$" cause desbordamiento aritmético sobre una máquina de 32 bits.

Advertencias VI

- Para guardar el resultado que corresponde a $n = 65535$ necesitaríamos 45496 bits, o más de 1420 palabras de computadora. Por la tanto, como un aspecto práctico no es realista considerar que estas operaciones pueden ejecutarse con costo unitario.

Advertencias VI

- Más bien, debemos atribuirles un costo proporcional a la longitud de los operandos involucrados. Posteriormente se mostrará que este algoritmo consume tiempo cuadrático, aunque a primera vista su tiempo de ejecución parece ser lineal.

Advertencias VII

- En el caso de la multiplicación aún podría ser razonable considerarla una operación elemental para operandos suficientemente pequeños.

Advertencias VII

- Desafortunadamente, es más fácil producir operandos grandes por multiplicaciones repetidas que por adiciones, por lo tanto es muy importante asegurar que las operaciones aritméticas no se desborden.

Advertencias VII

- Aún más, el tiempo requerido para efectuar una adición crece linealmente con respecto al tamaño de los operandos, pero el tiempo requerido para efectuar una multiplicación crece más rápido que eso.

Advertencias VIII

- Un problema similar puede ocurrir cuando analizamos algoritmos que involucran números reales si la precisión requerida aumenta con el tamaño de los ejemplares a resolver.

Advertencias VIII

- Un ejemplo típico de este fenómeno es cuando se usa la fórmula de De Moivre para calcular los valores de la secuencia de Fibonacci.

Advertencias VIII

- Esta fórmula nos dice que f_n el $n-$ésimo término en la secuencia, es aproximadamente igual a $\phi^n/\sqrt{5}$ donde $\phi = (1+\sqrt{5})/2$ es la **proporción dorada**.

Advertencias VIII

- La aproximación es tan suficientemente buena, que en principio podemos obtener el valor exacto de f_n tomando simplemente el entero más cercano.

Advertencias VIII

- No obstante, vimos que se necesitan alrededor de 45496 bits para representar exactamente a f_{65535}.

Advertencias IX

- Esto significa que tendríamos que calcular la aproximación con el mismo grado de exactitud que el requerido para obtener al número exacto.

Advertencias IX

- La aritmética de punto flotante de precisión simple o doble, usando una o dos palabras de computadora, ciertamente no sería suficientemente exacta.

Advertencias IX

- En la mayoría de las situaciones prácticas, no obstante, el uso de aritmética de punto flotante de precisión simple o doble ha sido satisfactoria, a pesar de la inevitable pérdida de precisión.

Advertencias IX

- Cuando esto es así, es razonable considerar dichas operaciones como de costo unitario.

Advertencias X

- Sumarizando, decidir cuando una instrucción tan aparentemente inofensiva como "$j \leftarrow i + j$" puede ser considerada como elemental o no, es algo que siempre requiere de nuestro juicio.

Advertencias X

- En lo que sigue, consideraremos como operaciones elementales a las adiciones, sustracciones, multiplicaciones, divisiones, operaciones módulo, operaciones Booleanas, comparaciones y asignamientos; y por lo tanto que pueden ejecutarse con costo unitario, al menos que explícitamente enunciemos lo contrario.

Ejercicios

Ejercicio 2

1. ¿Es razonable, como asunto práctico, considerar a la división como una operación elemental: (a) siempre, (b) algunas veces, (c) nunca? Justifique su respuesta. Si lo considera necesario, puede tratar separadamente la división de enteros y la división de números reales.
2. Implemente la función FibIter.
3. Implemente la función recursiva FibRec dada por la siguiente definición:

$$fibb(n) = \begin{cases} 0 & \text{cuando } n = 0, \\ 1 & \text{cuando } n = 1, \\ fibb(n-1) + fibb(n-2) & \text{cuando } n \geq 2. \end{cases}$$

4. Calcule FibRec(65535) y FibIter(65535).