Instituto Politécnico Nacional
Escuela Superior de Cómputo
Análisis de Algoritmos

Tarea 04

Complejidad de los algoritmos

Alumno:
López Manríquez Ángel (2017630941)

M. en C. Edgardo Adrián Franco Martínez

Grupo: 3CM3
Fecha: 22 de junio de 2019

Índice

1. Complejidad temporal

Supongamos los siguientes costos: c_1, c_2, c_3, c_4 para la asignacion, comparacion, operacion aritmetica y salto del contador del programa, respectivamente.

1.1. Algoritmo 1

para el primer y segundo `for` tenemos que $l(n) = (n-1)(c_1+c_2+c_4)+(n-2)(c_3+instrucciones\ internas)$. por tanto

```
for (int i = 1; i < n; i++)                  // (n - 1)(c1 + c2 + c4) + c3 (n - 2)
    for (int j = 0; j < n - 1; j++) {       // (n - 2)((n - 1)(c1 + c2 + c4) + c3 (n - 2))
        tmp = a[j];                          // c1 (n - 2) ^ 2
        a[j] = a[j + 1];                    // (c1 + c3)(n - 2) ^ 2
        a[j + 1] = tmp;                      // (c1 + c3)(n - 2) ^ 2
    }

```

suponiendo que $c_1 = c_2 = c_3 = c_4 = 1$ tenemos

$$\begin{aligned} f_t(n) &= 3(n-1) + (n-2) + 2(n-2)n - 1) + 5(n-2)^2 \\ &= 3(n-1) + (n-2)(1 + 3n - 3 + 5n - 12) \\ &= 9n^2 - 29n + 25 \text{si n} > 1 \end{aligned}$$

cuando $n \leq 1$ solo tenemos dos costos, la asignacion $i = 1$ y su comparacion $i < n$.

facilmente podemos ver que la complejidad es $O(n^2)$ una de las multiples implementaciones del ordenamiento por burbuja.

1.2. Algoritmo 2

Aqui se nos pide evaluar la regla de Horner para la evaluacion del un polinomio

```
polynomial = 0;                              // c1
for (int i = 0; i <= n; i++)                 // (c1 + c2 + c4)(n + 1) + c3 n
   polynomial = polynomial * z + a[n - 1]; // n (c1 + 2c3)

```

Claramente $O(n+1) = O(n)$ pues $i = 0, 1, ..., n+1$ cosa benefica pues la evaluacion de polinomios es lineal, la regla de Hurner es muy util pues la evaluacion es relativamente rapida a comparacion de la clasica donde se tiene que calcular la potencia, multiplicar y sumar.

1.3. Algoritmo 3

Para la evaluacion de un producto de matrices cuadradas de orden n, procedemos de manera analoga.

```
15 for (int i = 1; i <= n; i++)                    // 3(n + 1) + n
16    for (int j = 1; j <= n; j++) {                 // n(4n + 3)
17       c[i][j] = 0;                             // n ^ 2
18       for (int k = 1; k <= n; k++)              // (4n + 3) n ^ 2
19          c[i][j] = c[i][j] + a[i][k] * b[k][j];   // 3 n ^ 3
20    }
21
```

Asi: $T_n = n(4n+3) + n^2 + (4n+3)n^2 + 3n^3 = 7n^3 + 8n^2 + 5n + 6$ por lo que su complejidad es $O(n^3)$ como se nos ha dicho. Podriamos mejorar la funcion usando *matrix chain multiplication* la cual tiene aproximadamente $O(n^{2.7})$ la cual se valora muchisimo si los ordenes de las matrices es alto.

1.4. Algoritmo 4

```
23 last = 1;                // c1
24 current = 1;             // c1
25 while (n > 2) {             // (n - 1)(c2 + c4)
26    aux = last + current;     // (n - 2)(c1 + c3)
27    last = current;           // (n - 2) c1
28    current = aux;          // (n - 2) c1
29    --n;                // (n - 2) (c1 + c3)
30 }
31
```

Vemos que para $n > 2$: $O(6n - 12 + 2n - 2) = O(8n - 12) = O(n)$, esto se puede comprobar facilmente pues $i = n, n-1, ..., 2$.

1.5. Algoritmo 5

```
39 j = 0; // c1
40 for (i = n - 1; i >= 0; i--) // c3 (n - 1) + n (c1 + c2 + c4)
41    s2[j] = s[i]; // c1(n - 1)
42    j++; // (c1 + c3) (n - 1)
43 for (i = 0; i < n; i++) // c3(n - 1) + n(c1 + c2 + c4)
44    s[i] = s2[i]; // c1(n - 1)
```

Se aprecia que tanto el primer bucle como el segundo tienen $O(n)$, la suma, por supuesto, dara la misma complejidad. La funcion realizada por el algoritmo es poner al reves un vector, no es muy efectiva pues requiere el doble de espacio pues requerimos el vector actual y uno auxiliar, ademas de usar dos bucles.

2. Analisis de casos

En este apartado usaremos como operaciones basica: asignacion y comparacion y asumiremos que el costo de calcularlas es la unidad.

2.1. Algoritmo 9

```
1  #include <vector>
2  using std::vector;
3
4  // |A| >= 2
5  double productOfTheTwoBiggest(vector<double> A, int n) {
6      double biggest1, biggest2; // 0, biggest1 > biggest2
7
8      // 1 comparacion y 2 asignaciones
9     if (A[1] > A[2]) {
10         biggest1 = A[1];
11       biggest2 = A[2];
12     } else {
13       biggest1 = A[2];
14       biggest2 = A[1];
15     }
16
17    int i = 3; // 1 asignacion
18
19     // si n == 2 solo habra una comparacion
20    while (i <= n) { // n - 2 comparaciones, las operaciones del cuerpo se ejecutan n - 3 veces
21
22       if(A[i] > biggest1) { // una comparacion
23          biggest2 = biggest1; // una asignacion
24          biggest1 = A[i]; // una asignacion
25        } else if (A[i] > biggest2) // una comparacion
26          biggest2 = A[i]; // una asignacion
27
28       i = i + 1; // 1 asignacion
29     }
30    return biggest1 * biggest2; // 0 (no consideramos operaciones aritmeticas)
31 }
```

Consideremos el peor caso el cual se da cuando A_{n-1}, A_{n-2} son los valores mas grandes del vector, el `if` de la linea 22 costaria 3 pues A_i sera mayor que el actual mas grande, lo que implica que se hacen dos asignaciones mas. asi:

$$T_n = 5 + (n-2) + 4n = 5n + 3$$

por tanto su complejidad es $O(n)$.

2.2. Algoritmo 10

```cpp
template<typename T>
void swapSort(vector<T> a) {
    int n = a.size();
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++)
            if (a[j] < a[i]) // c1
                swap(a[j], a[i]); // c2
    }
}
```

Si consideramos el caso caso donde nos dan un vector ordenado al reves tenemos que el intercambio de elementos siempre se va a cumplir, al ser ambos bucles lineals donde cada uno tiene $O(n)$ y estar anidados tenemos que la complejidad del ordenamiento burbuja es $O(n^2)$.

2.3. Algoritmo 11

```python
def gcd(m, n):
   a = max(n, m)
   b = min(n, m)
   r = 1
   while b > 0:
      r = a % b
      a = b
      b = r
   return a
```

Usaremos como operación básica el módulo de `a` y `b`. Sin pérdida de generalidad asumamos que $m \geq n$, pues si $m < n$, se intercambian en las líneas 2 y 3. Ya habiamos llegado en el analisis anterior que $T_n \approx \log_\phi(n)$. Como al obtener la complejidad O mayuscula no nos interesan las bases del algoritmo tenemos $O(\log(n))$.

2.4. Algoritmo 12

```cpp
template<typename T>
void bubbleSortOptimized(std::vector<T> a, int n) {
    bool change = false;
    int i = 0;
    while (i < n - 1 && change != false) {
        change = false;
        for (int j = 0; j <= n - 2 - i)
            if (a[i] < a[j]) { // c1
                swap(a[i], a[j]); // c2
                change = true; // 1
            }
        ++i;
    }
}
```

Este algoritmo tiene la ventaja de que nos salimos del bucle si y solo si para algun k en adelante los elementos del vector estan ordenados. Como nos concentramos en el peor caso vemos que este algoritmo tiene la misma complejidad que el anterior, es decir $O(n^2)$.

2.5. Algoritmo 13

```cpp
#include <vector>
#include <iostream>

using std::vector;
using std::cout;

template <typename T>
void simpleBubble(vector<T> &a) {
    int n = a.size();
    for (int i = 0; i <= n - 2; i++)
        for (int j = 0; j <= n - 2 - i; j++) {
            if (a[j + 1] < a[j]) // c1
                swap(a[j], a[j + 1]); // c2
        }
}
```

Como de costumbre, enfocandonos en el peor caso tenemos que c_2 siempre se ejecuta, por tanto

$$\begin{aligned} T_n &= \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} (c_1 + c_2) \\ &= (c_1 + c_2) \sum_{i=0}^{n-2} (n - 2 - i)) \\ &= (c_1 + c_2) \left[n(n-1) - 2(n-1) - \frac{(n-2)(n-1)}{2} \right] \\ &= \frac{c_1 + c_2}{2} (n-1)(n-2) \end{aligned}$$

dando lugar a una $O(n^2)$. Como conclusion para cualquier implementacion de ordenamiento burbuja la complejidad gran O es cuadrada.

2.6. Algoritmo 14

```python
def sort(a, b, c):
   if a > b:
      if a > c:
         if b > c:
            output(a, b, c) # 3 pasos: a > b, a > c, b > c
         else:
            output(a, c, b)   # 3 pasos: a > b, a > c, b <= c
      else:
```

```
9           output(c, a, b)      # 2 pasos: a > b, a <= c
10      else:
11          if b > c:
12              if a > c:
13                  output(b, a, c)   # 3 pasos: a <= b, b > c, a > c
14              else:
15                  output(b, c, a)   # 3 pasos: a <= b, b > c, a <= c
16          else:
17              output(c, b, a)      # 2 pasos: a <= b, b <= c
```

Usaremos como operaciones básicas las comparaciones entre a, b y c.

Vemos que no hay ni bucles ni llamadas recursivas, si suponemos que la complejidad de output es constante y considerando el peor caso el cual se da para:

- $a > b$, $a > c$ y $b > c$
- $a > b$, $a > c$ y $b \leq c$
- $a \leq b$, $b > c$ y $a > c$
- $a \leq b$, $b > c$ y $a \leq c$

Dando como complejidad $f_t(n) = 3$. Vemos que el problema casi no depende de las variables del problema, dando como resultado $O(k)$.

2.7. Algoritmo 15

```
1  template<typename T>
2  void selection(vector<T> a, int n) {
3      for (int i = 0; i <= n - 2; i++) {
4          int minpos = i; // 1
5          for (int j = i + 1; j <= n -1; j++)
6              if (a[j] < a[minpos]) p = j; // c1
7          swap(a[p], a[i]); // c2
8      }
9  }
```

Este algoritmo no varia mucho en casos, mas sin embargo reduce el no. de intercambio de elementos, lo cual es muy conveniente si lo que se va a intercambiar es costoso en si. Se dice que la complejidad de este algoritmo es $O(n^2)$, lo cual es cierto pues para el peor caso tenemos:

Tenemos:

$$T_n = \sum_{i=0}^{n-2} \left(\sum_{j=i+1}^{n-1} c_1 + 1 + c_2 \right)$$
$$T_n = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} c_1 + (n-1)(c_2+1)$$
$$T_n = c_1 \sum_{i=0}^{n-2} (n-2-i-1+1) + (n-1)(c_2+1)$$
$$T_n = c_1 \frac{(n-2)(n-1)}{2} + (n-1)(c_2+1)$$
$$T_n = \frac{c_1}{2} n^2 + (c_2 - \frac{3}{2} c_1 + 1)n + c_1 - c_2 - 1$$

Vemos que, en efecto, la complejidad es $O(n^2)$.

Como una breve conclusion determinamos que las tecnicas vistas en clase si determinaron exitosamente cada una de las complejidades de la gran O pues al comparar con la tarea numero dos llegamos a los mismos resultados.