95.12 Algoritmos y Programación II

Práctica 5: análisis de algoritmos

Notas preliminares

- Esta práctica extiende la anterior otorgándole un marco práctico a la teoría básica de complejidad algorítmica y recurrencias.
- Los ejercicios marcados con el símbolo ♣ constituyen un subconjunto mínimo de ejercitación. No obstante, recomendamos fuertemente realizar todos los ejercicios.

---

1. Análisis de algoritmos iterativos

Ejercicio 1

Determinar una expresión Θ para el tiempo de corrida de peor caso de cada uno de los siguientes fragmentos de código.

(a)
```
f(n, 10, 0);
g(n, m, k);
h(n, m, 1000000);
```

(b)
```
for (int i = 0; i < n; ++i)
        f(n, m, k);
```

(c)
```
for (int i = 0; i < e(n, 10, 100); ++i)
        f(n, 10, 0);
```

(d)
```
for (int i = 0; i < e(n, m, k); ++i)
        f(n, 10, 0);
```

(e)
```
for (int i = 0; i < n; ++i)
for (int j = i; j < n; ++j)
        f(n, m, k);
```

Suponer que n, m y k son de tipo entero, y que las funciones e, f, g y h tienen las siguientes características:

- e(n, m, k) es O(1) y devuelve valores entre 1 y (n + m + k);
- f(n, m, k) es $O(n+m)$;
- g(n, m, k) es $O(m+k)$;
- h(n, m, k) es $O(n+k)$.

Ejercicio 2

Para cada uno de las siguientes funciones C++, averiguar qué es lo que calculan. Expresar la respuesta como función de n. Expresar el tiempo de corrida de peor caso en notación Θ.

(a)
```
int f(int n)
{
        int sum = 0;
        for (int i = 1; i <= n; ++i)
```

```
            sum += i;
        return sum;
}
```

(b)
```
int g(int n)
{
        int sum = 0;
        for (int i = 1; i < n; ++i)
                sum += i + f(n);
        return sum;
}
```

(c)
```
int h(int n)
{
        return f(n) + g(n);
}
```

Ejercicio 3 ♣

Para cada uno de los siguientes fragmentos de programa, encontrar la complejidad temporal expresada en notación Θ.

(a)
```
for (i = n; i >= 1; i /= 2)
        cout << "Awesome\n";
```

(b)
```
for (i = 1; i < n; i *= 2)
        for (j = 1; j <= i; ++j)
                cout << "Homework\n";
```

(c)
```
for (i = 1; i * i <= n; ++i)
        for (j = i; j >= 1; --j)
                cout << "Assignment\n";
```

Ejercicio 4 ♣

La función bubble_sort mostrada a continuación implementa un algoritmo de ordenamiento sobre listas:

```
template<class T>
void bubble_sort(list<T> &l)
{
    size_t n = l.length();

    for (size_t i = 1; i < n; ++i)
    for (size_t j = 0; j < n - i; ++j) {
            if (l[j] > l[j + 1])
                l.swap(j, j + 1);
    }
}
```

```
template<class T>
T &list<T>::operator[](size_t i)
{
    typename list<T>::iterator iter = this->begin();

    while (i--)
        ++iter;

    return *iter;
}
```

Siendo n la cantidad de elementos de la lista `l`,

a) Encontrar una expresión exacta en términos de n para la máxima cantidad de intercambios que puede realizar `bubble_sort`.

b) Dar una expresión asintótica en función de n para representar el tiempo de ejecución de peor caso de `bubble_sort`. ¿Creés que se trata de un algoritmo eficiente para ordenar listas?

c) Proponer un cambio en la función `bubble_sort` de forma tal de mejorar su complejidad asintótica. Luego de este cambio, ¿creés que se trata de un algoritmo eficiente para ordenar listas?

Ejercicio 5 ♣

La función `first_sort` mostrada a continuación implementa un algoritmo de ordenamiento sobre listas:

```
template<class T>
void first_sort(list<T> &l)
{
    typename list<T>::iterator it;
    T elem;

    while((it = first_unordered(l)) != l.end())
    {
        elem = *(++it);
        l.erase(it);
        l.push_front(elem);
    }
}
```

```
template<class T>
typename list<T>::iterator first_unordered(list<T> &l)
{
    typename list<T>::iterator prev = l.begin();
    typename list<T>::iterator next = l.begin();

    for(++next;  next != l.end(); ++prev, ++next)
        if(*prev > *next)
            return prev;

    return next;
}
```

Siendo n la cantidad de elementos de la lista `l`,

a) Dar una expresión asintótica en función de n para representar el tiempo de ejecución de `first_unordered`.

b) Encontrar una expresión exacta en términos de n para la máxima cantidad de intercambios que puede realizar `first_sort`.

c) Dar una expresión asintótica en función de n para representar el tiempo de ejecución de `first_sort`. ¿Creés que se trata de un algoritmo eficiente para ordenar listas?

2. Análisis de algoritmos recursivos

Ejercicio 6 ♣

El siguiente método C++ permite calcular la FFT de un arreglo de complejos cuyo tamaño es potencia entera de 2:

```
array<complex> &
fft::fftr(array<complex> const &src, array<complex> &dst, complex wbase) const
{
        size_t n = src.size();
        size_t m = src.size() / 2;

        if (n >= 2) {
                complex w(1, 0);
                array<complex> one(m);
                array<complex> two(m);

                // Si se trata de la primera instancia de la recursión,
                // inicializamos el factor exponencial a exp(-2 * pi * j / n),
                // en donde n representa el tamaño la longitud de arreglo a
```

```
                // transformar.
                //
                if (wbase == 0)
                        wbase = exp().conjugado();

                // Separamos el arreglo de entrada en 2 mitades, las de índice
                // par en el arreglo one, y las impares en two.
                //
                for (size_t k = 0; k < m; ++k) {
                        one[k] = src[2 * k];
                        two[k] = src[2 * k + 1];
                }

                // Calculamos recursivamente la transformada de ambas midates,
                // notando que el mismo código sirve para transformar en ambos
                // sentidos, directo e inverso: sólo es necesario cambiar el
                // signo del exponente del factor wbase.
                //
                fftr(one, one, wbase * wbase);
                fftr(two, two, wbase * wbase);

                // Combinamos ambas mitades para resolver la FFT solicitada.
                //
                for (size_t k = 0; k < n; ++k) {
                        dst[k] = one[k % m]
                               + two[k % m] * w;
                        w *= wbase;
                }
        } else {
                dst = src;
        }

        return dst;
}
```

(a) Dibujar el árbol de recursividad asociado a un arreglo de entrada de tamaño 4. Sólo es necesario mostrar los tamaños de los arreglos involucrados sin importar su contenido.

(b) Calcular la complejidad temporal.

(c) Calcular la complejidad espacial.

Ejercicio 7

El algoritmo Selección, delineado en la figura de abajo, permite obtener el i-ésimo menor elemento de un arreglo $A[1 \dots n]$ arbitrario, $n > 0$:

> Selección($A[1 \dots n]$, i):
> Si $n = 1$, devolver $A[1]$ (i debe ser 1).
> Dividir los elementos de A en grupos de 3 elementos.
> Sea $B[1 \dots n/3]$ un arreglo tal que $B[j]$ es la mediana del j-ésimo grupo.
> Sea $x :=$ Selección(B, $n/6$) la mediana de las medianas.
> Sea $A_1[1 \dots k]$ un arreglo con los elementos e de A tales que $e \leq x$.
> Sea $A_2[1 \dots n-k-1]$ un arreglo con los elementos e de A tales que $e > x$.
> Si $i = k + 1$, devolver x.
> Si $i \leq k$, devolver Selección(A_1, i).
> Si $i > k$, devolver Selección(A_2, $i - (k+1)$).

Dar una expresión asintótica usando notación Θ para representar la complejidad temporal de peor caso de SELECCIÓN.

Ejercicio 8

Supongamos una escalara de `dist` escalones, en donde sólo se permite subirla dando pasos o *steps* de a 1 escalón por vez, o saltando de a 2 escalones (*hops*). El siguiente pseudocódigo puede ser usado para calcular la cantidad de formas posibles de subir una escalera de tamaño `dist`:

```
void
climb_stairs(size_t dist, vector<string> &path = vector<string> ())
{
	if (dist >= 2) {
		path.push_back(string("hop"));
		climb_stairs(dist - 2, path);
		path.pop_back();
	}

	if (dist >= 1) {
		path.push_back(string("step"));
		climb_stairs(dist - 1, path);
		path.pop_back();
	}

	if (dist == 0) {
		const char *sep = "";

		for (size_t i = 0; i < path.size(); ++i) {
			cout << sep << path[i];
			sep = ", ";
		}
		cout << "\n";
	}
}
```

(a) Dibujar el árbol de recursividad asociado a `dist = 3`. ¿Qué imprime `climb_stairs(3)`?

(b) Caracterizar la complejidad temporal.

(c) Calcular la complejidad espacial.

(d) Comentar sobre la eficiencia de esta solución. ¿Cuáles de (b) y (c) podrían llegar a ser limitantes en una implementación que corra en una computadora relativamente moderna?

Ejercicio 9 ♣

Supongamos una función `lg_table(n)` que permite calcular el logaritmo de `n` en tiempo constante para valores del argumento menores o iguales a 2, y la siguiente implementación de `lg(n)`:

```
double
lg(double n)
{
	if (n <= 2)
		return lg_table(n);

	return 1 + lg(n / 2);
}
```

(a) Encontrar una expresión asintótica para la complejidad temporal de `lg(n)`.

(b) Caracterizar la complejidad espacial.

Supongamos ahora la siguiente implementación:

```
double
lg(double n)
{
        if (n <= 2)
                return lg_table(n);

        return 2 * lg(sqrt(n));
}
```

(c) Calcular una expresión asintótica para la complejidad temporal de esta función.

(d) Caracterizar la complejidad espacial.

Por último, consideremos:

```
double
lg(double n)
{
        if (n <= 2)
                return lg_table(n);

        return 1 + 2 * lg(sqrt(n / 2));
}
```

(e) Encontrar una expresión asintótica para la complejidad temporal de esta función. ¿Cuál de las tres implementaciones es más eficiente?

(f) Caracterizar la complejidad espacial.