Pauta Prueba 2

Analisis de Algoritmos (INF-648)
Primer Semestre 2012

Ejercicio 1

Supongamos que se debe elegir entre tres algoritmos:

- El algoritmo A resuelve un problema de tamaño n dividiendolo en 5 subproblemas de tamaño $\frac{n}{2}$, recursivamente resolviendo dichos problemas, y combinando las soluciones en tiempo lineal.
- El algoritmo B resuelve un problema de tamaño n resolviendo recursivamente 2 problemas de tamaño $n-1$ y luego combinando las soluciones en tiempo constante.
- El algoritmo C resuelve un problema de tamaño n dividiendolo en 9 subproblemas de tamaño $\frac{n}{3}$, recursivamente resolviendo dichos problemas, y combinando las soluciones en tiempo cuadratico.

Calcule el tiempo de ejecucion (osea, la complejidad computacional) de cada algoritmo, y indique cual algoritmo elegiria para resolver el problema.

Solucion

Algoritmo A

El algoritmo A posee una complejidad computacional que satisface la ecuacion de recurrencia:

$$T(n) = 5T\left(\frac{n}{2}\right) + n$$

Aplicando el teorema maestro con $a = 5$, $b = 2$, $f(n) = n$ y $E = log_b a = log_2 5 \sim 2.322$.

Chequeemos el Caso 1:

$$f(n) \in O(n^{E-\epsilon}), \quad \epsilon > 0$$

Si esto es cierto, entonces:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c, \qquad 0 \leq c < \infty$$

Desarrollando el limite y aplicando la Ley de L'Hopital:

$$\lim_{n\to\infty} \frac{n}{n^{E-\epsilon}} = \lim_{n\to\infty} n^{1-E+\epsilon}$$

Para que este limite tienda a cero, se debe cumplir que $1 - E + \epsilon < 0$, osea $1 - 2.322 + \epsilon < 0$, lo cual se cumple con $\epsilon < 1.322$. Entonces, este caso se cumple y por ende:

$$T(n) \in \Theta(n^E) = \Theta(n^{2.32}) \textbf{ 0.5 pts}$$

Algoritmo B

El algoritmo B posee una complejidad computacional que satisface la ecuacion de recurrencia:

$$T(n) = 2T(n-1) + c$$

Resolviendo esta ecuacion de recurrencia usando sustitucion hacia atras:

$$T(n) = 2T(n-1) + c$$
$$T(n-1) = 2T(n-2) + c$$

$$T(n) = 2^2T(n-2) + 2c + c$$
$$T(n-2) = 2T(n-3) + c$$

$$T(n) = 2^3T(n-3) + 2^2c + 2c + c$$
$$T(n-3) = 2T(n-4) + c$$

$$T(n) = 2^4T(n-4) + 2^3c + 2^2c + 2c + c$$

El patron es:

$$T(n) = 2^kT(n-k) + c\sum_{i=0}^{k-1} 2^i$$

Suponiendo que $T(0) = 1$:

$$T(n) = 2^n + c\sum_{i=0}^{n-1} 2^i = 2^n + c2^n \in \Theta(2^n)$$ **0.5 pts**

Algoritmo C

El algoritmo C posee una complejidad computacional que satisface la ecuacion de recurrencia:

$$T(n) = 9T\left(\frac{n}{3}\right) + n^2$$

Aplicando el teorema maestro con $a = 9$, $b = 3$, $f(n) = n^2$ y $E = log_b a = log_3 9 = 2$.

Chequeemos el Caso 1:

$$f(n) \in O(n^{E-\epsilon}), \quad \epsilon > 0$$

Si esto es cierto, entonces:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c, \qquad 0 \leq c < \infty$$

$$\lim_{n\to\infty} \frac{n^2}{n^{2-\epsilon}} = \lim_{n\to\infty} n^\epsilon$$

Para que este limite tienda a cero, se debe cumplir que $\epsilon < 0$, lo cual es una contradiccion al requisito de que $\epsilon > 0$, por ende la condicion no se cumple y este caso no aplica. Chequeemos el Caso 2:

$$f(n) \in \Theta(n^E \log^k n), \quad k \geq 0$$

Si esto es cierto, entonces:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c, \qquad 0 < c < \infty$$

Seleccionemos $k = 0$:

$$\lim_{n\to\infty} \frac{n^2}{n^2} = 1$$

Esta condicion se cumple y por ende:

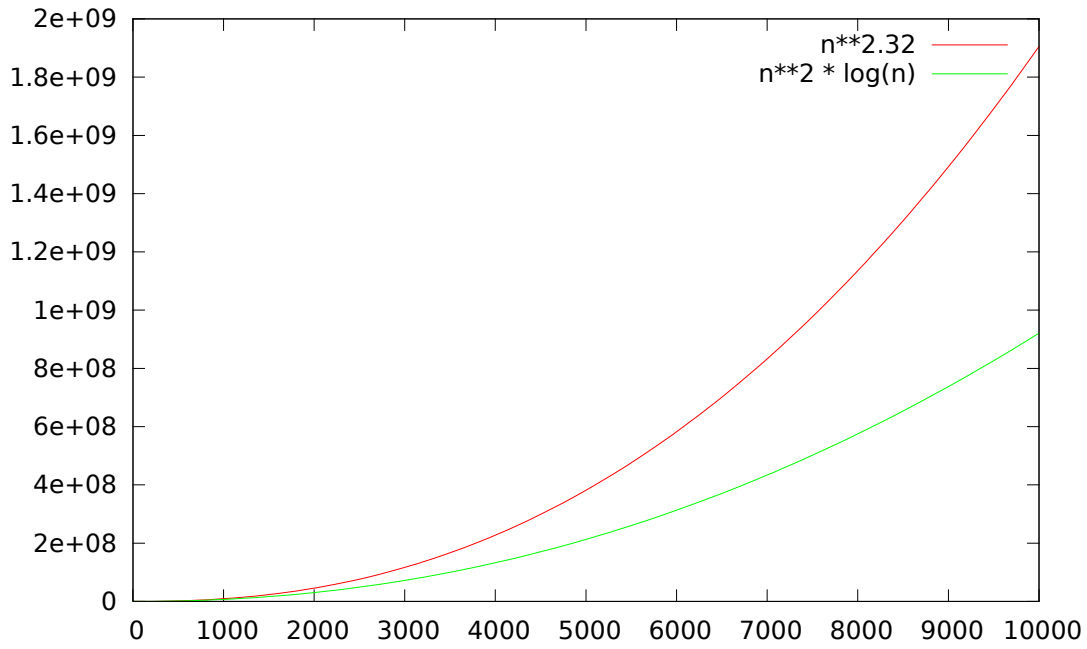
$$T(n) \in \Theta(n^E \log^{k+1} n) = \Theta(n^2 \log n)$$ **0.5 pts**

Decision

Las complejidades de cada algoritmo son:

- Algoritmo A: $\Theta(n^{2.32})$.
- Algoritmo B: $\Theta(2^n)$.
- Algoritmo C: $\Theta(n^2 \log n)$

Es obvio que el algoritmo mas rapido no es el algoritmo B, por ende se debe determinar el ganador entre el algoritmo A y el C. Un grafico de ambas funciones es util para determinar cual tiene una menor complejidad:



Con dicho grafico, podemos concluir que el algoritmo de menor complejidad computacional es el C. **0.5 pts**

Ejercicio 2

Construya un algoritmo "Divide-y-vencerás" que permita calcular los elementos mínimo y máximo de un arreglo de N números enteros, al mismo tiempo, donde N es una potencia de 2. Calcule la complejidad de su algoritmo.

Solucion

```
class Par
{
    public:
        Par(int min, int max) : minimo(min), maximo(max)
        {
        }

        int minimo;
        int maximo;
};

int max(int a, int b)
{
    return (a > b) ? a : b;
}
```

```
int min(int a, int b)
{
    return (a < b) ? a : b;
}

Par minMaxDYV(int *v, int i, int j)
{
    if(i < j) {

        int medio = (i + j) / 2;

        Par a = maxDYV(v, i, medio);
        Par b = maxDYV(v, medio + 1, j);

        return Par(min(a.min, b.min), max(a.max, b.max));

    } else {

        return Par(v[i], v[i]);
    }
}

Par minimoMaximo(int *v, int n)
{
    return minMaxDYV(a, 0, n - 1);
}
```

1.0 pts
La complejidad de este algoritmo esta dada por la ecuacion de recurrencia:

$$T(n) = 2T\left(\frac{n}{2}\right) + 1$$

Segun el teorema maestro, esta ecuacion tiene solucion:

$$T(n) \in \Theta(n) \textbf{ 0.5 pts}$$

Ejercicio 3

Construya un algoritmo recursivo que use programación dinámica para calcular a^n, con n un numero entero positivo y a un numero real.

Solucion

```
float powPD(float a, int n, float *almacen)
{
    if(almacen[n] != 0.0f) {
        return almacen[n];
    }

    float tmp;

    if(n % 2 == 0) {
        tmp = powPD(a, n / 2, almacen);
        almacen[n] = tmp * tmp;
    } else {
        tmp = powPD(a, n / 2, almacen);
        almacen[n] = tmp * tmp * a;
    }

    return almacen[n];
}
```

```
float pow(float a, int n)
{
    float *almacen = new int[n+1];

    for(int i = 0; i <= n; i++) {
        almacen[i] = 0.0f;
    }

    almacen[0] = 1.0f;
    almacen[1] = a;

    float ret = powPD(a, n, almacen);

    delete [] almacen;

    return ret;
}
```

1.5 pts

Ejercicio 4-a

Argumente porque $P \subset NP$.

Solucion

P es el conjunto de problemas que poseen algoritmos que los resuelven en tiempo polinomial. NP es el conjunto de problemas cuyas soluciones pueden ser chequeadas en tiempo polinomial. $P \subset NP$ es cierto, debido a que cualquier problema que esta en P, tambien esta en NP, si consideramos que construimo un algoritmo que en lugar de verificar la solucion, este la calcule, y como dicho costo es polinomial, dicho algoritmo tambien esta en NP.

Ejercicio 4-b

Si suponemos que P = NP, cual seria la complejidad de un algoritmo que resuelva el problema del vendedor viajero?

Solucion

El PVV es un problema NP-completo, lo que implica que todos los problemas NP se pueden reducir en tiempo polinomial a dicho problema. Si suponemos que P = NP, esto implica que todos los problemas en NP poseen una solucion en tiempo polinomial.

Entonces, como podemos reducir cualquier problema NP al problema del vendedor viajero en tiempo polinomial, el PVV tendria complejidad $O(n^p + n^k)$, donde p es el exponente de la reduccion, y k el exponente del algoritmo al cual se redujo, y dicha complejidad tambien es polinomial.

Por ende, si P = NP, el PVV tendria complejidad polinomial.