Algoritmos y Estructuras de Datos II, 2º del Grado en Ingeniería Informática

Test de Análisis de Algoritmos, marzo 2023 - miércoles - grupo 2.1

Vamos a ver la solución de los ejercicios detalladamente, con el objetivo de que se puedan entender bien las soluciones. En el examen se pide que se razonen las soluciones, aunque no es necesario que sea de forma tan detallada como vamos a verlo aquí. En los ejercicios de conteo de instrucciones vamos a contar el número total de instrucciones, incluyendo las comprobaciones de los bucles. También serían válidas otras opciones, como asignar constantes a las instrucciones, contar cada línea de código como 1 instrucción, contar o no contar las condiciones de los while, etc. Estas opciones pueden cambiar las constantes, aunque los órdenes de complejidad deben ser los mismos (excepto la o-pequeña, que sí depende de las constantes multiplicativas).

1) (3 puntos)

Podemos ver que el algoritmo a analizar consta de 3 bucles while anidados. El primer while tiene la forma:

```
i = 1
while i<=n
    …
    i++
endwhile
```

Por lo tanto, es evidente que este bucle se ejecutará siempre n veces. Es equivalente a un for i = 1…n. El segundo bucle tiene la forma:

```
k= i+1
while k<=n and condición
   …
   k++
endwhile
```

En este caso existen dos posibilidades. Si la condición es falsa, el bucle no se ejecutará ninguna vez. Si es cierta siempre, el bucle se ejecutará desde i+1 hasta n (se puede sustituir por un for k = i+1…n). Puesto que no sabemos a priori si la condición será cierta o falsa, debemos establecer un caso mejor (cuando la condición sea falsa y el bucle nunca se ejecuta) y un caso peor (cuando la condición sea siempre cierta y se ejecutan todas las iteraciones).

En el if index < M[k,i], la condición puede cumplirse o no. Pero tanto en un caso como en otro, siempre se ejecutará la comprobación del if y luego una asignación, es decir, 2 instrucciones siempre.

En cuanto al tercer bucle anidado, tiene la forma:

```
j= 1
while j <= n
   …
   j= j*2
endwhile
```

Como el índice j va aumentando de 2 en 2, en cada iteración r, la variable j valdrá 2^r. El bucle acabará cuando $2^r > n$, es decir, cuando $r > \log_2 n$. Así que el número de repeticiones sería $\lfloor \log_2 n \rfloor + 1$. Observar que si n = 1, el bucle se ejecuta 1 vez; si n = 2, el bucle se ejecuta 2 veces; si n = 4, el bucle se ejecuta 3 veces; etc. Por eso aparece un +1 en $\lfloor \log_2 n \rfloor + 1$.

Finalmente, el if de ese bucle más interno hace una comprobación que no podemos predecir a priori: if (M[k,j]>aux) … por lo que también podemos establecer un mejor y peor caso (el mejor es que no se cumpla, y el peor caso que se cumpla la condición). No obstante, sabemos que en el mejor caso del algoritmo nunca llegaremos a este bucle más interno, por lo que el mejor caso de este if es irrelevante.

Uniendo todo esto, podemos definir los tiempos en los casos mejor y peor, t_m y t_M, respectivamente. Vamos a estimar el tiempo contando el número de instrucciones ejecutadas. Aunque algunas instrucciones están agrupadas en una sola línea, las contaremos como dos instrucciones. En cuanto a los bucles while, tendremos en cuenta la propia comprobación del while, que siempre se realiza 1 vez más que el número de repeticiones del bucle (por ejemplo, aunque el bucle se ejecute 0 veces, siempre se ejecuta por lo menos la primera comprobación de la condición).

El tiempo en el mejor caso, $t_m(n)$, ocurre como hemos visto cuando en el segundo bucle while no se cumple la condición, de forma que tenemos:

$t_m(n)$ = 2 (asignaciones iniciales) + 1 (comprobación inicial del while) + n (repeticiones del primer while) * 5 (instrucciones internas del primer while, considerando las comprobaciones de los while) = 3 + 5n

Este tiempo en el mejor caso, $t_m(n)$, pertenece claramente a $\Theta(n)$. Como es el mejor caso del algoritmo, lo que podemos decir del tiempo en todos los casos, t(n), es que $t(n) \in \Omega(n)$.

En cuanto al tiempo en el peor caso, $t_M(n)$, se dará cuando las condiciones del segundo bucle y los if den lugar al mayor número de ejecuciones. Con lo que hemos visto arriba, sería:

$t_M(n)$ = 2 (asignaciones iniciales) + 1 (comprobación inicial del while) + n (repeticiones del primer while) * (5 (instrucciones internas del primer while, considerando las comprobaciones de los while) + (n-i) (repeticiones del segundo while en el peor caso) * (6 (instrucciones internas del segundo while, también contando las comprobaciones del segundo y tercer while) + ($\lfloor \log_2 n \rfloor$ + 1) (repeticiones del tercer while) * 4 (instrucciones internas del tercer while, suponiendo que el if se cumple siempre)))

La anterior fórmula sirve para explicar de dónde sale cada parte. Pero como el número de iteraciones del segundo while depende del índice i del primer while, no podemos simplemente multiplicar los valores anteriores (como hemos hecho en el mejor caso). Si consideramos los bucles while como si fueran bucles for, podemos expresar el conteo mediante tres sumatorios anidados (uno por cada while). Es decir:

$$t_M(n) = 3 + \sum_{i=1}^{n} \left(5 + \sum_{k=i+1}^{n} \left(6 + \sum_{r=1}^{log_2 n+1} 4\right)\right) =$$
$$= 3 + \sum_{i=1}^{n} \left(5 + \sum_{k=i+1}^{n} (6 + 4 \cdot (log_2 n + 1))\right) =$$
$$= 3 + \sum_{i=1}^{n} \left(5 + (n-i) \cdot (6 + 4 \cdot (log_2 n + 1))\right) =$$
$$= 3 + 5n + \sum_{i=1}^{n} \left((n-i) \cdot (10 + 4 \cdot log_2 n)\right) =$$
$$= 3 + 5n + (10 + 4 \cdot log_2 n) \cdot \sum_{i=1}^{n} (n-i) =$$

Para seguir simplificando esta fórmula, vamos a ver cuánto vale:

$$\sum_{i=1}^{n} (n-i) = n^2 - \sum_{i=1}^{n} i = \left(n^2 - \frac{n \cdot (n+1)}{2}\right) = n^2 - \frac{n^2}{2} - \frac{n}{2} = \frac{n^2}{2} - \frac{n}{2}$$

Así que seguimos simplificando $t_M(n)$:

$$t_M(n) = 3 + 5n + \left(\frac{n^2}{2} - \frac{n}{2}\right)(10 + 4 \cdot log_2 n) =$$
$$= 3 + 5n + (n^2 - n)(5 + 2 \cdot log_2 n) =$$
$$= 3 + 5n + 5n^2 + 2n^2 \cdot log_2 n - 5n - 2n \cdot log_2 n =$$
$$= 3 + 5n^2 + 2n^2 \cdot log_2 n - 2n \cdot log_2 n$$

Como sabemos, el orden de complejidad de una suma (tanto O, como Ω y Θ) es igual al orden del máximo. Por lo tanto, podemos decir que $t_M(n) \in \Theta(n^2 \cdot \log n)$. Y como este es el peor caso del algoritmo, sobre el tiempo t en todos los casos podemos decir que $t(n) \in O(n^2 \cdot \log n)$.

En definitiva, como tenemos que $t(n) \in \Omega(n)$ y $t(n) \in O(n^2 \cdot \log n)$, no podemos dar un orden exacto, Θ, para el tiempo de ejecución del algoritmo en todos los casos.

2) (2 puntos)

a) $t(n) = 3t(n-1) + 4t(n-2) + n3^n + n^3 + 3^n$

Aplicando el método de la ecuación característica, tenemos el siguiente desarrollo:

- **Primer paso:** pasar los términos en t al lado izquierdo, y los que no están en t al lado derecho.

$$t(n) - 3t(n-1) - 4t(n-2) = n3^n + n^3 + 3^n$$

- **Segundo paso**: para la parte izquierda, transformar los términos en t en potencias de x:

$$(x^2 - 3x - 4) = 0$$

Y resolvemos la ecuación:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} = \frac{3 \pm \sqrt{3^2 + 4 \cdot 4}}{2} = \frac{3 \pm \sqrt{9 + 16}}{2} = \frac{3 \pm \sqrt{25}}{2} = \frac{3 \pm 5}{2} = \left(\frac{8}{2}, -\frac{2}{2}\right) = (4, -1)$$

- **Tercer paso**: para la parte derecha, buscamos cosas de la forma $b^n \cdot$polinomio(n), que dan lugar a la solución b, con multiplicidad el grado del polinomio + 1. Debemos agrupar todos los términos que tengan la misma base del exponente; en este caso, debemos agrupar $n \cdot 3^n$ y 3^n.

$$(n+1) \cdot 3^n + n^3 \cdot 1^n \rightarrow x = (3, 3, 1, 1, 1, 1)$$

- **Cuarto paso**: con todas las soluciones, x= (4, -1, 3, 3, 1, 1, 1, 1), construimos la fórmula expandida para t(n). Omitimos aquí escribir las potencias 1^n, puesto que siempre valen 1.

$$t(n) = c_1 \cdot 4^n + c_2 \cdot (-1)^n + c_3 \cdot 3^n + c_4 \cdot n \cdot 3^n + c_5 + c_6 \cdot n + c_7 \cdot n^2 + c_8 \cdot n^3$$

Como no sabemos los casos base, no se pueden resolver las constantes c_i. Podemos decir que $t(n) \in O(4^n)$. También podríamos decir que $t(n) \in \Theta(4^n)$, aunque esto solo lo podemos asegurar si sabemos que la constante c_1 es distinta de 0.

b) $t(n) = 3t(n-1) + 4t(n-2) + n^3 + n^2 4^{n/2} + 1$

- **Primer paso**: pasar los términos en t al lado izquierdo, y los que no están en t al lado derecho.

$$t(n) - 3t(n-1) - 4t(n-2) = n^3 + n^2 4^{n/2} + 1$$

- **Segundo paso**: en este caso, podemos ver que los términos que están en t son los mismos que en el apartado a). Por lo tanto, tenemos la misma ecuación característica ($x^2 – 3x – 4 = 0$), cuyas soluciones son x = (4, -1).

- **Tercer paso**: para los términos que no están en t, en este caso agrupamos ($n^3 + 1$) (solución 1 con multiplicidad 4) y por otro lado $4^{n/2}\cdot(n^2)$ (solución $4^{1/2}$ con multiplicidad 3). Pero $4^{1/2}$ es lo mismo que la raíz cuadrada de 4, es decir 2. Por lo tanto tenemos las soluciones: x = (1, 1, 1, 1, 2, 2, 2).

- **Cuarto paso**: las soluciones son: x= (4, -1, 1, 1, 1, 1, 2, 2, 2), construimos la fórmula expandida para t(n).

$$t(n) = c_1\cdot 4^n + c_2\cdot(-1)^n + c_3 + c_4\cdot n + c_5\cdot n^2 + c_6\cdot n^3 + c_7\cdot 2^n + c_8\cdot n\cdot 2^n + c_9\cdot n^2\cdot 2^n$$

En cuanto al orden de complejidad, el mayor término entre 4^n, n^3 y $n^2\cdot 2^n$ es 4^n, por lo que también tenemos que $t(n) \in O(4^n)$. Y, como antes, para dar un orden exacto deberíamos saber que c_1 no vale 0.

3) (1 punto)

Vamos a ver primero lo que ocurre con cada uno de estos términos:

- $O(2^{0.1^n})$. En general, los exponenciales (b^n) son funciones que crecen muy rápidamente. Pero, ¿qué pasa cuando la base del exponente es menor que 1, como 0.1^n, cuando n tiende a infinito? Por ejemplo, $0.1^2 = 0.01$; $0.1^3 = 0.001$; $0.1^4 = 0.0001$. Cuando n tiende a infinito, 0.1^n tiende a 0. Por lo tanto, 2 elevado a una cosa que tiende a 0, tenderá a 1 cuando n tiende a infinito. Así que $2^{0.1^n}$ es realmente una función decreciente, que tiende a 1. De hecho, por este razonamiento, podemos decir que $O(2^{0.1^n}) = O(1)$.
- $O(n^3 ln^2 n + (\sqrt{n})^5)$. Sabemos que el orden de una suma es igual al orden del máximo de los términos (y lo mismo con el Ω y el Θ). El término $n^3\cdot \ln^2 n$ no se puede simplificar. El segundo término es equivalente a: $(n^{1/2})^5 = n^{5/2} = n^{2.5}$. Por lo tanto, el primer término es mayor, teniendo que $O(n^3 ln^2 n + (\sqrt{n})^5) = O(n^3 ln^2 n)$.
- $O(9^{n/2})$. Podemos simplificar el exponencial con: $9^{n/2} = (9^{1/2})^n = 3^n$. Así que este término vale $O(3^n)$.
- $O(\log_{10}(n^2)\cdot \ln(n)\cdot \log_3 n)$. Sabemos que los logaritmos con diferentes bases (en base 10, en base 3, en base e), solo se diferencian en una constante. Por lo tanto: $\log_{10}(n^2)\cdot \ln(n)\cdot \log_3 n = C\cdot \log(n^2)\cdot \log(n)\cdot \log(n)$. Y el $\log(n^2)$ es lo mismo que $2\cdot \log(n)$. Así que el orden simplificado es $O(\log n\cdot \log n\cdot \log n) = O((\log n)^3) = O(\log^3 n)$.
- $O(\log_3 n^{5/2})$. Igual que antes, tanto la base del logaritmo (3) como el exponente de n (5/2), solo van a afectar a una constante multiplicativa, por lo que $O(\log_3 n^{5/2}) = O(\log n)$.

Juntando todo lo anterior, podemos ordenar los O de la siguiente forma:

$$O(2^{0.1^n}) \subset O(\log_3 n^{5/2}) \subset O(\log_{10}(n^2)\cdot \ln(n)\cdot \log_3 n) \subset O(n^3 ln^2 n + (\sqrt{n})^5) \subset O(9^{n/2})$$

Con los Ω, el orden de la inclusión es el contrario:

$$\Omega(2^{0.1^n}) \supset \Omega(\log_3 n^{5/2}) \supset \Omega(\log_{10}(n^2)\cdot \ln(n)\cdot \log_3 n) \supset \Omega(n^3 ln^2 n + (\sqrt{n})^5) \supset \Omega(9^{n/2})$$

Sin embargo, con los Θ no podemos establecer ninguna relación de inclusión. Simplemente podemos decir que los Θ de todos ellos son distintos entre sí.

En este ejercicio, las relaciones que aparecen con los O y con los Ω son siempre inclusiones. Pero también podría haber ejemplos donde existan relaciones de igualdad. Por ejemplo, suponer que en este ejercicios añadimos los órdenes: $O(7)$, $O(\log_{10} n)$. Entonces, las relaciones entre los órdenes serían:

$$O(2^{0.1^n}) = O(7) \subset O(\log_3 n^{5/2}) = O(\log_{10} n) \subset O(\log_{10}(n^2)\cdot \ln(n)\cdot \log_3 n) \subset \dots$$

4) (1 punto)

a) Si $t_M(n) \in \Theta(n^n)$ y $t_m(n) \in \Omega(2^n)$ entonces $t(n) \in \Theta(n!)$

La afirmación es falsa. Si el tiempo en el peor caso, $t_M(n)$, pertenece al orden exacto $\Theta(n^n)$, significa que la ejecución más lenta del algoritmo tarda un orden de complejidad de $\Theta(n^n)$. Puesto que t(n) incluye todos los posibles casos (el peor, el mejor y todos los casos intermedios), entonces también incluye el caso dado por $t_M(n)$. Por lo tanto, si $t_M(n) \in \Theta(n^n)$, entonces solo existen dos posibilidades para el orden exacto de t(n): (i) que $t(n) \in \Theta(n^n)$; o (ii) que no exista un orden exacto para t(n). Así que no es posible que t(n) pertenezca a un orden exacto distinto. Lo que podríamos decir es que $t(n) \in O(n^n)$.

Por otro lado, $t_m(n) \in \Omega(2^n)$ significa que el tiempo en el mejor caso es de un orden 2^n o mayor. De esta forma, $t_m(n)$ puede ser por ejemplo 2^n, n! o n^n porque todos ellos están acotados inferiormente por $\Omega(2^n)$. Pero si el tiempo es precisamente 2^n, entonces t(n) no puede ser un orden exacto de $\Theta(n!)$.

b) $t_m(n) \in \Omega(t_p(n))$ implica $t_M(n) \in O(t_p(n))$

La afirmación es falsa. Sabemos que el tiempo en el mejor caso, t_m, siempre será menor o igual que el tiempo en el peor caso, t_M. Y el tiempo promedio, t_p, siempre estará entre ambos, pudiendo ser igual a uno de ellos. Es decir, siempre se cumple que: $t_m(n) \leq t_p(n) \leq t_M(n)$. Traducido a órdenes de complejidad, esto significa que $O(t_m(n)) \subseteq O(t_p(n)) \subseteq O(t_M(n))$. Por ello, el hecho de que $t_m(n) \in \Omega(t_p(n))$ implica que $O(t_m(n)) = O(t_p(n))$. Es decir, el caso mejor tiene el mismo orden de complejidad que el caso promedio. Pero esto no significa que el caso peor tenga también el mismo orden de complejidad que el promedio (que es lo que significaría $t_M(n) \in O(t_p(n))$.

Por ejemplo, supongamos que tenemos un algoritmo donde el mejor caso es un O(n), el caso promedio es un O(n) y el peor caso es un $O(n^2)$. Entonces, es cierto que $t_m(n) \in \Omega(t_p(n))$, pero no es cierto que $t_M(n) \in O(t_p(n))$.

5) (3 puntos)

Vamos a hacer primero el conteo de instrucciones del algoritmo. Como en el primer ejercicio, vamos a contar todas las instrucciones y comprobaciones de los bucles y los if, aunque estén en la misma línea. Realmente este algoritmo siempre va a devolver valor 0, puesto que el valor de cont nunca deja de valer 0. Así que el compilador podría optimizar el código para que tarde siempre un O(1). Pero vamos a suponer que no es así.

Por otro lado, aunque el algoritmo recibe el tamaño en el parámetro m, se nos dice que la llamada inicial es con g21(P, Q, n). Por lo tanto, vamos a suponer que la m que recibe el algoritmo vale n. Puesto que las llamadas recursivas van dividiendo por 2 y por 4, vamos a suponer que n es potencia de 2. Es decir, $n = 2^k$, siendo k un entero. Luego, $k = \log_2 n$.

En cuanto al bucle while, tiene una forma típica de empezar el índice i en 1, termina en n, y va aumentando de 2 en 2. Como vimos en el primer ejercicio, esto da lugar a que el número de repeticiones sea $(\log_2 n) + 1$.

En el caso promedio, suponemos que la probabilidad de que cierto número de un array sea par es 1/2, y la probabilidad de que sea impar es también 1/2. Como la condición if (par(P[m]) OR impar(Q[m])) será cierta tanto se P[m] es par, como si Q[m] es impar, entonces la probabilidad de que se cumpla es 3/4 (de las 4 posibles combinaciones par/impar, en 3 casos el if será cierto y solo en 1 será falso). Juntando todo esto, tenemos el siguiente resultado para el conteo de instrucciones en el caso promedio (vamos a poner t en lugar de t_p, por simplificar un poco la notación):

$t(n) = 5 + 2 \cdot (1 + \log_2 n) = 7 + 2 \cdot \log_2 n$ Si $n \leq 8$

$t(n) = 5 + 2 \cdot (1 + \log_2 n) + 2 + 3/4 \cdot 4 \cdot (2 + t(n/2)) + 1/4 \cdot 16 \cdot (2 + t(n/4))$ Si $n > 8$

Simplificando el caso general tenemos:

$$t(n) = 5 + 2 + 2 \cdot \log_2 n + 2 + 6 + 3 \cdot t(n/2) + 8 + 4 \cdot t(n/4) = 23 + 2 \cdot \log_2 n + 3 \cdot t(n/2) + 4 \cdot t(n/4)$$

Para poder resolverla por el método de la ecuación característica, necesitamos primero hacer un cambio de variable: $n = 2^k$, luego $k = \log^2 n$.

$$t(2^k) = 23 + 2 \cdot k + 3 \cdot t(2^{k-1}) + 4 \cdot t(2^{k-2})$$

Tomamos la función $t'(k) = t(2^k)$. Luego nos queda:

$$t'(k) = 23 + 2 \cdot k + 3 \cdot t'(k-1) + 4 \cdot t'(k-2)$$

Sobre esta ecuación podemos aplicar el método de la ecuación característica, puesto que ya tenemos las recursividades restando, en lugar de dividiendo. Por lo tanto, en primer lugar pasamos los términos en t' a la parte izquierda:

$$t'(k) - 3 \cdot t'(k-1) - 4 \cdot t'(k-2) = 23 + 2 \cdot k$$

Considerando la parte izquierda, aparece la ecuación característica: $x^2 - 3x - 4 = 0$. Curiosamente es la misma ecuación que en el ejercicio 2a), para la que ya vimos que sus soluciones son x = (4, -1).

Para la parte derecha, vemos que hay un polinomio en k de grado 1, luego aparecen las soluciones: x= (1, 1). En total, las soluciones para x son (4, -1, 1, 1). Y la expansión para t'(k):

$$t'(k) = c_1 \cdot 4^k + c_2 \cdot (-1)^k + c_3 + c_4 \cdot k$$

Ahora podemos deshacer el cambio de variable para obtener el resultado en n, sabiendo que $t'(k) = t(2^k) = t(n)$, y también que $k = \log_2 n$.

$$t(n) = c_1 \cdot 4^{\log_2 n} + c_2 \cdot (-1)^{\log_2 n} + c_3 + c_4 \cdot \log_2 n$$

Debemos recordar que: $a^{\log_b c} = c^{\log_b a}$. Por lo tanto, $4^{\log_2 n} = n^{\log_2 4} = n^2$. Así que nos queda:

$$t(n) = c_1 \cdot n^2 + c_2 \cdot (-1)^{\log_2 n} + c_3 + c_4 \cdot \log_2 n$$

Para resolver las constantes c_1, c_2, c_3 y c_4, necesitamos 4 condiciones iniciales, que nos darán 4 ecuaciones. Estas 4 condiciones iniciales deben cumplir la condición de que n sea potencia de 2 y, además, deben ser casos alcanzables desde el caso general. Como el caso general se aplica para n > 8, el primer valor para que el que se aplica el caso general es t(16), cuyo valor sería:

$$t(16) = 23 + 2 \cdot \log_2 16 + 3 \cdot t(16/2) + 4 \cdot t(16/4) = 23 + 2 \cdot 4 + 3 \cdot t(8) + 4 \cdot t(4)$$

Esto nos hace ver que los casos base t(2) y t(1) no son alcanzables desde el caso general. Por lo tanto, para resolver las constantes, debemos aplicar las condiciones iniciales: t(4), t(8), t(16) y t(32). Los casos menores o iguales a 8 los obtenemos de los casos base de la ecuación ($t(n) = 7 + 2 \cdot \log_2 n$):

- $t(4) = 7 + 2 \cdot \log_2 4 = 7 + 2 \cdot 2 = 11 = c_1 \cdot 4^2 + c_2 \cdot (-1)^{\log_2 4} + c_3 + c_4 \cdot \log_2 4$
  $11 = 16 \cdot c_1 + c_2 + c_3 + 2 \cdot c_4$
- $t(8) = 7 + 2 \cdot \log_2 8 = 7 + 2 \cdot 3 = 13 = c_1 \cdot 8^2 + c_2 \cdot (-1)^{\log_2 8} + c_3 + c_4 \cdot \log_2 8$
  $13 = 64 \cdot c_1 - c_2 + c_3 + 3 \cdot c_4$

Los casos para n mayor que 8 los obtenemos de la definición recurrente ($t(n)= 23 + 2 \cdot \log_2 n + 3 \cdot t(n/2) + 4 \cdot t(n/4)$). Obtenemos las ecuaciones:

- $t(16) = 23 + 2 \cdot \log_2 16 + 3 \cdot t(8) + 4 \cdot t(4) = 23 + 2 \cdot 4 + 3 \cdot 13 + 4 \cdot 11 = 114 = c_1 \cdot 16^2 + c_2 \cdot (-1)^{\log_2 16} + c_3 + c_4 \cdot \log_2 16$
  $114 = 256 \cdot c_1 + c_2 + c_3 + 4 \cdot c_4$

- $t(32) = 23 + 2\cdot\log_2 32 + 3\cdot t(16) + 4\cdot t(8) = 23 + 2\cdot 5 + 3\cdot 114 + 4\cdot 13 = 427 = c_1\cdot 32^2 + c_2\cdot(-1)^{\log_2 32} + c_3 + c_4\cdot\log_2 32$
  $427 = 1024\cdot c_1 - c_2 + c_3 + 5\cdot c_4$

Es complicado resolver manualmente este sistema de ecuaciones (lo cual no se pide en el examen). Resolviéndolo con software, obtenemos: $c_1 = 0{,}432$; $c_2 = 9{,}2$; $c_3 = -4{,}444$; $c_4 = -0{,}333$. En consecuencia, el tiempo en el caso promedio sería (aquí vamos a volver a poner $t_p(n)$ para dejar claro que el resultado obtenido es sobre el caso promedio):

$$t_p(n) = 0{,}432\cdot n^2 + 9{,}2\cdot(-1)^{\log_2 n} - 4{,}444 - 0{,}333\cdot\log_2 n$$

En cuanto a los órdenes de complejidad, sin resolver las constantes podemos decir que $t_p(n) \in O(n^2 \mid n=2^k)$. Si sabemos o asumimos que la constante c_1 es positiva y distinta de 0, entonces también podríamos decir que $t_p(n) \in \Omega(n^2 \mid n=2^k)$ y $t_p(n) \in \Theta(n^2 \mid n=2^k)$. Y también $t_p(n) \in o(c_1\cdot n^2 \mid n=2^k)$. Podríamos intuir que c_1 no será 0, puesto que si fuera 0 entonces el tiempo sería un $O(\log_2 n)$, lo que significaría que las llamadas recursivas que aparecen no hacen que aumente el tiempo de ejecución. Por otro lado, c_1 no puede ser negativa, porque entonces el tiempo sería negativo para tamaños grandes, lo cual nunca puede ocurrir (ojo, una ecuación de recurrencia general puede producir valores negativos, pero una ecuación de recurrencia que se deriva de un conteo de instrucciones nunca puede producir valores negativos).

Finalmente, para eliminar la condición en los órdenes condicionados, debemos demostrar que $t_p(n)$ es eventualmente no decreciente, que n^2 es eventualmente no decreciente y que n^2 es 2-armónica. La función $t_p(n)$ será eventualmente no decreciente si la constante c_1 es positiva, como hemos visto que ocurre (o bien lo suponemos si no la calculamos). Por otro lado, es trivial que n^2 es una función creciente y, por lo tanto, es también eventualmente no decreciente. Y también se cumple que n^2 es 2-armónica, puesto que se cumple que $(2n)^2 = 4n^2 \in \Theta(n^2)$. Todas las funciones polinomiales son 2 armónicas (por ejemplo, 1; log n; n; n^2; n^3; etc.) pero las exponenciales no lo son (por ejemplo, 2^n, 3^n, n!, n^n, etc.). Por lo tanto, como se cumplen las 3 condiciones, podemos quitar los tres órdenes condicionados y asegurar que: $t_p(n) \in O(n^2)$, $t_p(n) \in \Omega(n^2)$ y $t_p(n) \in \Theta(n^2)$. Ojo, el teorema visto en clase no nos permite quitar el orden condicionado sobre la o-pequeña.