La complejidad temporal en el mejor de los casos...

Seleccione una:

a. ... es el tiempo que tarda el algoritmo en resolver la talla más pequeña que se le puede presentar.

b. ... es una función de la talla que tiene que estar definida para todos los posibles valores de ésta.

c. Las demás opciones son verdaderas.

Un algoritmo recursivo basado en el esquema *divide y vencerás*...

Seleccione una:

a. Las demás opciones son verdaderas.

b. ... será más eficiente cuanto más equitativa sea la división en subproblemas.

c. … nunca tendrá una complejidad exponencial.

La versión de *Quicksort* que utiliza como pivote la mediana del vector...

Seleccione una:

a. ... no presenta caso mejor y peor distintos para instancias del mismo tamaño.

b. ... es más eficiente si el vector ya está ordenado.

c. ... es la versión con mejor complejidad en el mejor de los casos.

La versión de *Quicksort* que utiliza como pivote el elemento del vector que ocupa la posición central...

Seleccione una:

a. ... se comporta mejor cuando el vector ya está ordenado.

b. ... se comporta peor cuando el vector ya está ordenado.

c. ... no presenta casos mejor y peor distintos para instancias del mismo tamaño.

La versión de *Quicksort* que utiliza como pivote el elemento del vector que ocupa la primera posición..

Seleccione una:

a. ... se comporta mejor cuando el vector ya está ordenado.

b. ... se comporta peor cuando el vector ya está ordenado.

c. ... El hecho de que el vector estuviera previamente ordenado o no, no influye en la complejidad temporal de este algoritmo.

Los algoritmos de ordenación *Quicksort y Mergesort* tienen en común ...

Seleccione una:

a. ... que se ejecutan en tiempo O(n).

b. ... que ordenan el vector sin usar espacio adicional.

c. ... que aplican la estrategia de *divide y vencerás*.

Sobre la complejidad temporal de la siguiente función:

```
unsigned desperdicio (unsigned n){
if (n<=1)
     return 0;
unsigned sum = desperdicio (n/2) + desperdicio (n/2) + desperdicio
(n/2);
for (unsigned i=1; i<n-1; i++)
    for (unsigned j=1; j<=i; j++)
        for (unsigned k=1; k<=j; k++)
            sum+=i*j*k;
return sum;
}
```

Seleccione una:

a. Ninguna de las otras dos alternativas es cierta.

b. Las complejidades en los casos mejor y peor son distintas.

c. El mejor de los casos se da cuando $n \leq 1$ y en tal caso la complejidad es constante.

Con respecto al esquema *Divide y vencerás*, ¿es cierta la siguiente afirmación?

Si la talla se reparte equitativamente entre los subproblemas, entonces la complejidad temporal resultante es una función logarítmica.

Seleccione una:

a. No, nunca, puesto que también hay que añadir el coste de la división en subproblemas y la posterior combinación.

b. No tiene porqué, la complejidad temporal no depende únicamente del tamaño resultante de los subproblemas.

c. Sí, siempre, en Divide y Vencerás la complejidad temporal depende únicamente del tamaño de los subproblemas.

¿Cuál de estas tres expresiones es falsa?

Seleccione una:

a. $2n^3 - 10n^2 + 1 \in O(n^3)$

b. $n + n\sqrt{n} \in \Omega(n)$

c. $n + n\sqrt{n} \in \Theta(n)$

Sea $f(n)=n\ log(n)+n$.

Seleccione una:

- a. ... $f(n)\in\Omega(n\ log(n))$
- b. ... $f(n)\in O(n\ log(n))$
- (•) **c. Las otras dos opciones son ciertas**

Si $f_1(n)\in O(g_1(n))$ **y** $f_2(n)\in O(g_2(n))$ **entonces...**

Seleccione una:

- (•) **a. Las otras dos alternativas son ciertas.**
- b. $f_1(n)+f_2(n)\in O(\max(g_1(n),g_2(n)))$
- c. $f_1(n)+f_2(n)\in O(g_1(n)+g_2(n))$

¿Cuál es la complejidad temporal de la siguiente función?

```
int ejemplo (vector < int > & v){

    int n=v.size();
    int j,i=2;
    int sum=0;
    while (n>0 && i<n){
        j=i;
        while (v[j] != v[1]){
            sum+=v[j];
            j=j/2;
        }
        i++;
    }
    return sum;
}
```

Seleccione una:

- (•) **a.** $O(n\log n)$
- b. $O(n^2)$
- c. $\Theta(n\log n)$

En cuanto a la complejidad temporal de la siguiente función:

```
int ejemplo (vector < int > & v){

    int n=v.size();
    int j,i=2;
    int sum=0;
    while (n>0 && i<n){
        j=i;
        while (v[j] != v[1]){
            sum+=v[j];
            j=j/2;
        }
        i++;
    }
    return sum;
}
```

Seleccione una:

a. Las complejidades en el mejor y en el peor de los casos no coinciden.

b. El mejor de los casos se da cuando $n = 0$, su complejidad es constante.

c. Esta función no presenta casos mejor y peor puesto que sólo puede haber una instancia para cada una de las posibles talla

Indica cuál es la complejidad, en función de n, del fragmento siguiente:

```
for( int i = n; i > 0; i /=2 )
    for(  int j = n; j > 0; j /=2 )
        a += A[i][j];
```

Seleccione una:

a. $O(log^2(n))$

b. $O(n \log(n))$

c. $O(n^2)$

Indica cuál es la complejidad, en función de n, del fragmento siguiente:

```
a = 0;
for( int i = 0; i < n*n; i++ )
    a += A[(i + j) % n];
```

Seleccione una:

a. $O(n^2)$

b. $O(n \log(n))$

c. $O(n)$

El siguiente fragmento del algoritmo de ordenación *Quicksort* reorganiza los elementos del vector para obtener una subsecuencia de elementos menores que el pivote y otra de mayores. Su complejidad temporal, con respecto al tamaño del vector `v`, que está delimitado por los valores `pi` y pf, es...

```
x = v[pi];
i = pi+1;
j = pf;
do {
      while( i<=pf && v[i] < x ) i++;
      while( v[j] > x ) j--;
      if( i <= j ) {
                     swap( v[i],v[j]);
                     i++;
                     j--;
      }
} while( i < j );
swap(v[pi],v[j]);
```

Nota: La función swap se realiza en tiempo constante.

Seleccione una:

- (•) **a. ... lineal en cualquier caso.**
- ( ) b. ... cuadrática en el peor de los casos.
- ( ) c. ... lineal en el caso peor y constante en el caso mejor.

Indica cuál es la complejidad de la función siguiente:

```
unsigned sum( const mat &A ) {      // A es una matriz cuadrada
unsigned d = A.n_rows();
unsigned a = 0;
   for( unsigned i = 0; i < d; i++ )
       for( unsigned j = 0; j < d; j++ )
           a += A(i,j);
return a;
}
```

Seleccione una:

- ( ) a. $O(n\log(n))$
- ( ) b. $O(n^2)$
- (•) **c.** $O(n)$

1 + f(n/3)	-> Θ(log(n))
1 + f(n/2)	-> Θ(log(n))
1 + 2f(n/2)	-> Θ(n)
sqrt(n) + 3f(n/3)	-> Θ(n)
1 + f(n - 1)	-> Θ(n)
n + 3f(n/3)	-> Θ(nlog(n))
n + 2f(n/2)	-> Θ(nlog(n))
n + f(n/2) + f(n/2)	-> Θ(nlog(n))
n + 4f(n / 2)	-> Θ(n^2)
n^2 + 3f(n / 3)	-> O(n^2)
n + f(n - 2)	-> O(n^2)
n^2 + 4f(n / 2)	-> O(n^2log2(n))
n + 2f(n - 1)	-> Ω(2^n)
1 + 2f(n - 1) -	> Θ(2^n)