Pregunta 1:   
  
Escribe la especificación formal de un TAD Natural que, además de las operaciones constructoras, disponga de las operaciones serPar, serImpar, suma, producto y factorial.  
  
  
  
  
Respuesta 1:   
  
Especificación formal del TAD Natural:  
  
Tipo de dato: Natural  
  
Operaciones:  
- crearNatural(n: entero): Natural  
 - Descripción: Crea un número natural a partir de un entero.  
 - Entrada: Un entero n.  
 - Salida: Un Natural.  
 - Precondición: n >= 0.  
 - Postcondición: El Natural creado tiene el valor n.  
  
- serPar(): booleano  
 - Descripción: Verifica si el número natural es par.  
 - Entrada: Ninguna.  
 - Salida: Un booleano que indica si el número es par (true) o no (false).  
 - Precondición: Ninguna.  
 - Postcondición: Ninguna.  
  
- serImpar(): booleano  
 - Descripción: Verifica si el número natural es impar.  
 - Entrada: Ninguna.  
 - Salida: Un booleano que indica si el número es impar (true) o no (false).  
 - Precondición: Ninguna.  
 - Postcondición: Ninguna.  
  
- suma(n: Natural): Natural  
 - Descripción: Realiza la suma del número natural con otro número natural.  
 - Entrada: Un Natural n.  
 - Salida: Un Natural que representa la suma de los dos números.  
 - Precondición: Ninguna.  
 - Postcondición: Ninguna.  
  
- producto(n: Natural): Natural  
 - Descripción: Realiza el producto del número natural con otro número natural.  
 - Entrada: Un Natural n.  
 - Salida: Un Natural que representa el producto de los dos números.  
 - Precondición: Ninguna.  
 - Postcondición: Ninguna.  
  
- factorial(): Natural  
 - Descripción: Calcula el factorial del número natural.  
 - Entrada: Ninguna.  
 - Salida: Un Natural que representa el factorial del número.  
 - Precondición: Ninguna.  
 - Postcondición: Ninguna.

Pregunta 2:   
  
Dada una función de Fibonacci implementada recursivamente, i) escribe su complejidad computacional temporal en el peor de los casos, en el mejor de los casos, y el caso promedio, ii) explica qué estrategia algorítmica sigue esa implementación y iii) si invocamos Fibonacci(5), ¿cuántas veces se llama la función a sí misma (di el número total y luego sepáralo por casos en función del parámetro de entrada de la función). Nota: no escribas la función, simplemente responde a los apartados anteriores.  
  
  
Respuesta 2:   
  
i) La complejidad computacional temporal en el peor de los casos de la implementación recursiva de la función de Fibonacci es exponencial, O(2^n), donde n es el parámetro de entrada de la función. Esto se debe a que en cada llamada recursiva se realizan dos llamadas adicionales, lo que lleva a un crecimiento exponencial del número de llamadas.  
  
En el mejor de los casos, la complejidad sería constante, O(1), cuando el parámetro de entrada es 0 o 1, ya que en esos casos no se realizan llamadas recursivas y se devuelve directamente el valor correspondiente.  
  
En el caso promedio, la complejidad también es exponencial, ya que la mayoría de los valores de Fibonacci requerirán un número exponencial de llamadas recursivas para calcularse.  
  
ii) La estrategia algorítmica seguida por esta implementación recursiva de Fibonacci es la de dividir el problema en subproblemas más pequeños y resolverlos recursivamente. En cada llamada recursiva, se calcula el valor de Fibonacci para los dos números anteriores y se suman para obtener el valor actual.  
  
iii) Si invocamos Fibonacci(5), la función se llamará a sí misma un total de 14 veces. Esto se puede desglosar de la siguiente manera:  
  
- Fibonacci(5) se llama a sí misma 9 veces.  
- Fibonacci(4) se llama a sí misma 5 veces.  
- Fibonacci(3) se llama a sí misma 3 veces.  
- Fibonacci(2) se llama a sí misma 2 veces.  
- Fibonacci(1) y Fibonacci(0) no se llaman a sí mismas.  
  
En resumen, Fibonacci(5) se llama a sí misma 14 veces en total.

Pregunta 3:   
  
El siguiente código muestra un TAD Lista con implementación doblemente enlazada.  
  
typedef int TIPOELEM;  
  
typedef struct celda {TIPOELEM elemento;  
  
 struct celda \* ant,sig;} TCELDA;  
  
typedef TCELDA \* POSICION;  
  
typedef struct lista {  
  
 int longitud;  
  
 POSICION fin;} \* TLISTA;  
  
i) Escribe la implementación en C de la función recupera  
  
void recupera(TLISTA l, POSICION p, TIPOELEM \*e)  
  
{  
  
}  
  
ii) Escribe la implementación en C de la función crea  
  
void crea(TLISTA \*l)  
  
{  
  
}  
  
iii) Escribe la complejidad computacional temporal en el peor de los casos de la función recupera  
  
iv) Escribe la complejidad computacional temporal en el peor de los casos de una función que destruyese toda la lista  
  
  
Respuesta 3:   
  
i) Implementación de la función recupera:  
  
void recupera(TLISTA l, POSICION p, TIPOELEM \*e)  
{  
 if (p != NULL) {  
 \*e = p->elemento;  
 }  
}  
  
ii) Implementación de la función crea:  
  
void crea(TLISTA \*l)  
{  
 \*l = (TLISTA)malloc(sizeof(struct lista));  
 (\*l)->longitud = 0;  
 (\*l)->fin = NULL;  
}  
  
iii) La complejidad computacional temporal en el peor de los casos de la función recupera es O(1), ya que simplemente se accede al elemento de la posición p.  
  
iv) La complejidad computacional temporal en el peor de los casos de una función que destruyese toda la lista sería O(n), donde n es la longitud de la lista. Esto se debe a que se tendría que recorrer toda la lista para liberar la memoria de cada celda.

Pregunta 4:   
  
En una lista queremos buscar 4 personas que cumplan los requisitos para configurar un tribunal. Cada persona tiene que tener al menos 10 años de experiencia profesional y, además, tiene que haber equilibrio de género en el tribunal generado (50% hombres y 50% mujeres). Explica qué complejidad temporal en el mejor y en el peor de los casos tendría una función que implementase ese proceso de la forma más eficiente posible. No escribas la función en C, simplemente explica qué complejidad temporal tendría en el mejor y en el peor de los casos.  
  
  
Respuesta 4:   
  
La complejidad temporal de una función que implemente el proceso de búsqueda de 4 personas que cumplan los requisitos y generen un tribunal equilibrado en términos de género dependerá de la forma en que se almacenen y se organicen los datos de las personas.  
  
En el mejor de los casos, si los datos de las personas están organizados de tal manera que se pueda acceder directamente a las personas que cumplen los requisitos y se pueda encontrar rápidamente un equilibrio de género, la complejidad temporal sería O(1). Esto significa que la función sería extremadamente eficiente y no importaría el tamaño de la lista de personas, ya que siempre se encontrarían las 4 personas adecuadas en un tiempo constante.  
  
Sin embargo, en el peor de los casos, si los datos de las personas no están organizados de manera óptima y se requiere recorrer toda la lista de personas para encontrar a las 4 adecuadas y lograr el equilibrio de género, la complejidad temporal sería O(n), donde n es el número de personas en la lista. Esto significa que la función sería menos eficiente y el tiempo requerido para encontrar las 4 personas adecuadas aumentaría linealmente con el tamaño de la lista de personas.  
  
En resumen, la complejidad temporal de la función que implemente el proceso de búsqueda de personas para configurar un tribunal equilibrado en términos de género puede variar desde O(1) en el mejor de los casos hasta O(n) en el peor de los casos, dependiendo de cómo se organicen y almacenen los datos de las personas.

Pregunta 5:   
  
Shin Chan es un díscolo e irreverente niño japonés interesado en las construcciones con piezas Lego. Quiere hacer una pirámide con piezas Lego 2x2. Cada planta de la pirámide va creciendo en tamaño, en el tope de la pirámide hay una única pieza, en el segundo nivel hay 4, en el tercer nivel hay 9 y así sucesivamente.  
  
La cuestión es que Shin Chan quiere saber cuántas piezas necesitaría para construir una pirámide de cualquier número de alturas (en cada planta el interior de la pirámide está totalmente lleno de piezas, no hay huecos).  
  
Shin Chan no está motivado para trabajar en resolver este problema, su único interés se centra en sus soeces actividades diarias. Por ello, contrata a Boo, una avezada programadora de aplicaciones, que escribe una función recursiva que, dado el número de alturas, determina el número total de piezas que Shin Chan necesita. Escribe la función que Boo crearía para ayudar a Shin Chan.  
  
  
Respuesta 5:   
  
La función recursiva que Boo crearía para ayudar a Shin Chan sería la siguiente:  
  
```python  
def calcular\_piezas\_piramide(altura):  
 if altura == 1:  
 return 1  
 else:  
 return altura\*\*2 + calcular\_piezas\_piramide(altura-1)  
```  
  
Explicación de la función:  
  
- La función `calcular\_piezas\_piramide` recibe como parámetro la altura de la pirámide.  
- Si la altura es igual a 1, significa que solo hay una planta en la pirámide, por lo que se necesita una única pieza. En este caso, la función retorna 1.  
- Si la altura es mayor a 1, se utiliza la fórmula `altura\*\*2` para calcular el número de piezas en la planta actual. Luego, se llama recursivamente a la función `calcular\_piezas\_piramide` con una altura menor en 1 para calcular el número de piezas en las plantas restantes. Finalmente, se suma el número de piezas en la planta actual con el número de piezas en las plantas restantes y se retorna el resultado.  
  
Ejemplo de uso de la función:  
  
```python  
altura\_piramide = 5  
total\_piezas = calcular\_piezas\_piramide(altura\_piramide)  
print("El número total de piezas necesarias para una pirámide de altura", altura\_piramide, "es:", total\_piezas)  
```  
  
Salida:  
```  
El número total de piezas necesarias para una pirámide de altura 5 es: 55  
```  
  
En este ejemplo, se calcula el número total de piezas necesarias para una pirámide de altura 5, y el resultado es 55.

Pregunta 6:   
  
Una institución está organizando un curso de formación para el que tiene sólo un determinado cupo de plazas. Recibe peticiones por orden de llegada pero sólo las personas que cumplen ciertos criterios (edad >18 y <65 y nota en un test previo de idoneidad >5) son admisibles. Escribe una función en C que reciba el cupo máximo de admisibles (número entero) y una cola donde se han almacenado las peticiones y escriba por pantalla las personas que pueden incorporarse al curso. Cada persona se representa por su nombre, DNI, edad y nota en el test de idoneidad. Antes de escribir la función escribe el TIPOELEMENTO que tendría el TAD Cola. Abajo figuran las especificaciones de las operaciones del TAD Cola que puedes usar.  
  
typedef void \* TCOLA;  
  
void ColaVacia (TCOLA \*q);  
  
void AnadirCola (TCOLA \*q, TELEMENTO e);  
  
int EsColaVacia (TCOLA q);  
  
void EliminarCola (TCOLA \*q);  
  
void PrimeroCola (TCOLA q , TELEMENTO \*e);  
  
  
  
  
Respuesta 6:   
  
El TIPOELEMENTO que tendría el TAD Cola en este caso sería una estructura que contenga los datos de una persona, como su nombre, DNI, edad y nota en el test de idoneidad. Por ejemplo:  
  
typedef struct {  
 char nombre[50];  
 char dni[10];  
 int edad;  
 float nota;  
} TELEMENTO;  
  
A continuación, se muestra la implementación de la función solicitada:  
  
void personasAdmisibles(int cupoMaximo, TCOLA \*cola) {  
 TCOLA colaAdmisibles;  
 ColaVacia(&colaAdmisibles);  
   
 while (!EsColaVacia(\*cola) && cupoMaximo > 0) {  
 TELEMENTO persona;  
 PrimeroCola(\*cola, &persona);  
   
 if (persona.edad > 18 && persona.edad < 65 && persona.nota > 5) {  
 AnadirCola(&colaAdmisibles, persona);  
 cupoMaximo--;  
 }  
   
 EliminarCola(cola);  
 }  
   
 while (!EsColaVacia(colaAdmisibles)) {  
 TELEMENTO persona;  
 PrimeroCola(colaAdmisibles, &persona);  
 printf("Nombre: %s\nDNI: %s\nEdad: %d\nNota: %.2f\n\n", persona.nombre, persona.dni, persona.edad, persona.nota);  
 EliminarCola(&colaAdmisibles);  
 }  
}  
  
En esta función, se crea una nueva cola llamada "colaAdmisibles" donde se almacenarán las personas que cumplen los criterios de admisión. Se recorre la cola original y se verifica si cada persona cumple los criterios. Si cumple, se añade a la cola de admisibles y se decrementa el cupo máximo. Luego, se eliminan las personas de la cola original.  
  
Finalmente, se recorre la cola de admisibles y se imprime por pantalla la información de cada persona.  
  
Es importante tener en cuenta que esta implementación asume que las funciones del TAD Cola están correctamente implementadas y que la cola original contiene elementos del tipo TELEMENTO.

Pregunta 7:   
  
El gráfico de debajo muestra la complejidad computacional de una solución divide y vencerás en función de los parámetros k, c e i.  
  
-Si k<c^i entonces t(n) ∈ O(n^i)  
  
-Si k=c^i entonces t(n) ∈ O(n^i log n)  
  
-Si k>c^i entonces t(n) ∈ O(n^(log\_c k))  
  
a) Explica el significado de k, c e i (indicando también si queremos que cada uno de ellos tenga un valor alto o bajo y por qué)  
  
b) En las dos variantes de multiplicación de enteros grandes (x multiplicado por y) descritas debajo, indica cual es más eficiente, justificando porqué en referencia a sus parámetros divide y vencerás.   
  
Variante 1  
  
xy=aci^n+(ad+bc)i^(n/2)+bd  
  
Variante 2  
  
xy=aci^n+((a-b)(d-c)+ac+bd)i^(n/2)+bd  
  
c) Compara esta complejidad computacional con la que tendríamos con una búsqueda secuencial con y sin centinela.  
  
  
Respuesta 7:   
  
a)   
  
- k: Es un parámetro que representa el tamaño del problema. Queremos que k sea alto, ya que esto implica que el problema es grande y, por lo tanto, más complejo.  
  
- c: Es un parámetro que representa la cantidad de subproblemas en cada división. Queremos que c sea bajo, ya que esto implica que cada división genera pocos subproblemas, lo que facilita la resolución del problema.  
  
- i: Es un parámetro que representa la cantidad de veces que se divide el problema original. Queremos que i sea alto, ya que esto implica que el problema se divide en muchas partes, lo que aumenta la complejidad.  
  
b)   
  
Para determinar cuál de las dos variantes de multiplicación de enteros grandes es más eficiente, debemos analizar la complejidad computacional de cada una de ellas.  
  
Variante 1: La complejidad computacional de esta variante es O(n^i), donde n es el tamaño del problema. Esto se debe a que la fórmula tiene tres términos, cada uno de ellos con una potencia de i. Por lo tanto, la complejidad aumenta exponencialmente con i.  
  
Variante 2: La complejidad computacional de esta variante es O(n^i log n), donde n es el tamaño del problema. Esto se debe a que la fórmula tiene tres términos, cada uno de ellos con una potencia de i, pero además se agrega un término logarítmico debido a la presencia de i^(n/2). Por lo tanto, la complejidad aumenta exponencialmente con i, pero también se agrega un factor logarítmico.  
  
En referencia a los parámetros divide y vencerás, la variante 1 es más eficiente que la variante 2. Esto se debe a que la variante 1 tiene una complejidad computacional más baja, ya que no incluye el factor logarítmico adicional.  
  
c)   
  
La complejidad computacional de una búsqueda secuencial sin centinela es O(n), donde n es el tamaño del arreglo en el que se realiza la búsqueda. Esto se debe a que en el peor caso, se deben recorrer todos los elementos del arreglo para encontrar el elemento buscado.  
  
La complejidad computacional de una búsqueda secuencial con centinela es O(n+1), donde n es el tamaño del arreglo en el que se realiza la búsqueda. Esto se debe a que se agrega un elemento adicional al arreglo (el centinela), lo que permite evitar la comprobación de límites en cada iteración del bucle. Sin embargo, en términos asintóticos, la complejidad sigue siendo O(n).  
  
En comparación con la complejidad computacional de la solución divide y vencerás, tanto la búsqueda secuencial sin centinela como la búsqueda secuencial con centinela tienen una complejidad computacional más baja. Esto se debe a que la complejidad de la solución divide y vencerás aumenta exponencialmente con i, mientras que la complejidad de las búsquedas secuenciales es lineal.