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:  
  
El TAD Natural representa un número natural y tiene las siguientes operaciones:  
  
- Constructoras:  
 - crearNatural(n: entero): crea un número natural a partir de un entero n.  
  
- Observadoras:  
 - serPar(n: Natural): devuelve verdadero si el número n es par, falso en caso contrario.  
 - serImpar(n: Natural): devuelve verdadero si el número n es impar, falso en caso contrario.  
  
- Operaciones aritméticas:  
 - suma(n1: Natural, n2: Natural): devuelve la suma de los números n1 y n2.  
 - producto(n1: Natural, n2: Natural): devuelve el producto de los números n1 y n2.  
 - factorial(n: Natural): devuelve el factorial del número n.  
  
La especificación formal del TAD Natural en lenguaje C sería la siguiente:  
  
'''  
typedef struct {  
 int valor;  
} Natural;  
  
Natural crearNatural(int n) {  
 Natural natural;  
 natural.valor = n;  
 return natural;  
}  
  
int serPar(Natural n) {  
 return n.valor % 2 == 0;  
}  
  
int serImpar(Natural n) {  
 return n.valor % 2 != 0;  
}  
  
Natural suma(Natural n1, Natural n2) {  
 Natural resultado;  
 resultado.valor = n1.valor + n2.valor;  
 return resultado;  
}  
  
Natural producto(Natural n1, Natural n2) {  
 Natural resultado;  
 resultado.valor = n1.valor \* n2.valor;  
 return resultado;  
}  
  
Natural factorial(Natural n) {  
 Natural resultado;  
 resultado.valor = 1;  
 for (int i = 1; i <= n.valor; i++) {  
 resultado.valor \*= i;  
 }  
 return resultado;  
}  
'''  
  
Esta especificación formal define el TAD Natural con las operaciones constructoras y las operaciones serPar, serImpar, suma, producto y factorial. Cada operación está implementada de acuerdo a su definición matemática correspondiente.>

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 de la implementación recursiva de la función de Fibonacci en el peor de los casos es exponencial, O(2^n), donde n es el número de términos de la secuencia de Fibonacci que se desea calcular. 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 a medida que n aumenta.  
  
En el mejor de los casos, la complejidad es constante, O(1), cuando se solicita el cálculo del primer término de la secuencia (Fibonacci(0) o Fibonacci(1)), ya que no se realizan llamadas recursivas adicionales.  
  
En el caso promedio, la complejidad es difícil de determinar exactamente, ya que depende de la distribución de los valores de entrada. Sin embargo, en general, la complejidad sigue siendo exponencial, aunque puede ser menor que en el peor de los casos si la secuencia de Fibonacci se calcula para valores más pequeños.  
  
ii) La estrategia algorítmica seguida por la implementación recursiva de la función de Fibonacci es la de dividir y conquistar. En cada llamada recursiva, se divide el problema en subproblemas más pequeños (los términos anteriores de la secuencia) y se resuelven de manera recursiva. Luego, los resultados de los subproblemas se combinan para obtener el resultado final.  
  
iii) Si invocamos Fibonacci(5), la función se llama a sí misma un total de 15 veces. Esto se puede desglosar en los siguientes casos:  
  
- 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) se llama a sí misma 1 vez.  
- Fibonacci(0) se llama a sí misma 0 veces.  
  
En general, el número de llamadas a sí misma de la función de Fibonacci sigue un patrón similar a la secuencia de Fibonacci misma, donde cada término es la suma de los dos términos anteriores.>

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 en C de la función recupera:  
  
```c  
void recupera(TLISTA l, POSICION p, TIPOELEM \*e)  
{  
 \*e = p->elemento;  
}  
```  
  
ii) Implementación en C de la función crea:  
  
```c  
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 y se asigna a la variable e.  
  
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 y liberar la memoria ocupada por 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 un algoritmo se refiere a la cantidad de tiempo que tarda en ejecutarse en función del tamaño de los datos de entrada. En este caso, estamos buscando 4 personas que cumplan ciertos requisitos en una lista de personas.  
  
La forma más eficiente de implementar este proceso sería recorrer la lista de personas una sola vez y realizar las comprobaciones necesarias para determinar si cada persona cumple con los requisitos. En el peor de los casos, tendríamos que recorrer toda la lista de personas para encontrar las 4 personas que cumplen los requisitos. Por lo tanto, la complejidad temporal en el peor de los casos sería O(n), donde n es el número de personas en la lista.  
  
En el mejor de los casos, podríamos encontrar las 4 personas que cumplen los requisitos en las primeras posiciones de la lista. En este caso, solo tendríamos que recorrer una pequeña parte de la lista antes de encontrar las 4 personas. Por lo tanto, la complejidad temporal en el mejor de los casos sería O(1), es decir, constante.  
  
En resumen, la complejidad temporal en el peor de los casos sería lineal (O(n)), mientras que en el mejor de los casos sería constante (O(1)). Esto significa que, en general, el algoritmo sería eficiente y tendría un buen rendimiento, especialmente si la lista de personas no es demasiado grande.>

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 siguiente función en C resuelve el problema planteado:  
  
```c  
#include <stdio.h>  
  
int calcularPiezas(int altura) {  
 // Caso base: si la altura es 1, solo se necesita una pieza  
 if (altura == 1) {  
 return 1;  
 }  
 // Caso recursivo: el número de piezas necesarias es igual a la altura al cuadrado más el número de piezas necesarias para la altura anterior  
 else {  
 return altura \* altura + calcularPiezas(altura - 1);  
 }  
}  
  
int main() {  
 int altura;  
 printf("Ingrese la altura de la pirámide: ");  
 scanf("%d", &altura);  
 int totalPiezas = calcularPiezas(altura);  
 printf("El número total de piezas necesarias es: %d\n", totalPiezas);  
 return 0;  
}  
```  
  
Explicación paso a paso:  
  
1. La función `calcularPiezas` toma como parámetro la altura de la pirámide y devuelve el número total de piezas necesarias.  
2. En el caso base, si la altura es igual a 1, solo se necesita una pieza, por lo que se devuelve 1.  
3. En el caso recursivo, el número de piezas necesarias es igual a la altura al cuadrado más el número de piezas necesarias para la altura anterior. Esto se logra llamando a la función `calcularPiezas` con la altura decrementada en 1.  
4. En la función `main`, se solicita al usuario que ingrese la altura de la pirámide.  
5. Se llama a la función `calcularPiezas` con la altura ingresada y se almacena el resultado en la variable `totalPiezas`.  
6. Se imprime en pantalla el número total de piezas necesarias.  
  
Esta función utiliza recursividad para calcular el número total de piezas necesarias para construir una pirámide de cualquier altura. Cada llamada recursiva reduce la altura en 1 hasta llegar al caso base, donde se devuelve 1. Luego, se suman los resultados de las llamadas recursivas para obtener el número total de piezas.>

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 tipo de elemento TAD Cola se puede definir de la siguiente manera:  
  
```c  
typedef struct {  
 char nombre[50];  
 char DNI[10];  
 int edad;  
 int nota;  
} TELEMENTO;  
```  
  
A continuación, se presenta la función en C que recibe el cupo máximo de admisibles y una cola de peticiones, y muestra por pantalla las personas que cumplen los criterios de admisión:  
  
```c  
#include <stdio.h>  
#include <stdlib.h>  
  
typedef struct {  
 char nombre[50];  
 char DNI[10];  
 int edad;  
 int nota;  
} TELEMENTO;  
  
typedef void \* TCOLA;  
  
void ColaVacia(TCOLA \*q) {  
 \*q = NULL;  
}  
  
void AnadirCola(TCOLA \*q, TELEMENTO e) {  
 TCOLA nuevo\_elemento = malloc(sizeof(struct Nodo));  
 nuevo\_elemento->elemento = e;  
 nuevo\_elemento->siguiente = NULL;  
  
 if (\*q == NULL) {  
 \*q = nuevo\_elemento;  
 } else {  
 TCOLA aux = \*q;  
 while (aux->siguiente != NULL) {  
 aux = aux->siguiente;  
 }  
 aux->siguiente = nuevo\_elemento;  
 }  
}  
  
int EsColaVacia(TCOLA q) {  
 return q == NULL;  
}  
  
void EliminarCola(TCOLA \*q) {  
 if (\*q != NULL) {  
 TCOLA aux = \*q;  
 \*q = (\*q)->siguiente;  
 free(aux);  
 }  
}  
  
void PrimeroCola(TCOLA q, TELEMENTO \*e) {  
 if (q != NULL) {  
 \*e = q->elemento;  
 }  
}  
  
void mostrarPersonasAdmisibles(int cupoMaximo, TCOLA peticiones) {  
 int contador = 0;  
 TELEMENTO persona;  
  
 while (!EsColaVacia(peticiones) && contador < cupoMaximo) {  
 PrimeroCola(peticiones, &persona);  
 if (persona.edad > 18 && persona.edad < 65 && persona.nota > 5) {  
 printf("Nombre: %s\n", persona.nombre);  
 printf("DNI: %s\n", persona.DNI);  
 printf("Edad: %d\n", persona.edad);  
 printf("Nota: %d\n", persona.nota);  
 contador++;  
 }  
 EliminarCola(&peticiones);  
 }  
}  
  
int main() {  
 TCOLA peticiones;  
 ColaVacia(&peticiones);  
  
 // Agregar peticiones a la cola (ejemplo)  
 TELEMENTO peticion1 = {"Juan", "12345678A", 25, 7};  
 TELEMENTO peticion2 = {"María", "87654321B", 30, 4};  
 TELEMENTO peticion3 = {"Pedro", "98765432C", 40, 8};  
  
 AnadirCola(&peticiones, peticion1);  
 AnadirCola(&peticiones, peticion2);  
 AnadirCola(&peticiones, peticion3);  
  
 int cupoMaximo = 2;  
 mostrarPersonasAdmisibles(cupoMaximo, peticiones);  
  
 return 0;  
}  
```  
  
En este ejemplo, se crea una cola de peticiones y se agregan algunas peticiones de ejemplo. Luego, se llama a la función `mostrarPersonasAdmisibles` pasando el cupo máximo y la cola de peticiones. La función recorre la cola y muestra por pantalla las personas que cumplen los criterios de admisión (edad > 18, edad < 65 y nota > 5) hasta alcanzar el cupo máximo.>

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) En el contexto de la solución divide y vencerás, los parámetros k, c e i tienen los siguientes significados:  
  
- k: Es un factor que determina la relación entre los subproblemas y el problema original. Si k es bajo, significa que los subproblemas son mucho más pequeños que el problema original, lo que implica que la solución divide y vencerás es más eficiente. Si k es alto, significa que los subproblemas son comparables en tamaño al problema original, lo que puede hacer que la solución divide y vencerás sea menos eficiente.  
  
- c: Es un factor que determina el tamaño de los subproblemas en cada nivel de recursión. Si c es bajo, significa que los subproblemas son mucho más pequeños en cada nivel de recursión, lo que implica que la solución divide y vencerás es más eficiente. Si c es alto, significa que los subproblemas son comparables en tamaño en cada nivel de recursión, lo que puede hacer que la solución divide y vencerás sea menos eficiente.  
  
- i: Es el exponente que determina cómo se divide el problema original en subproblemas más pequeños. Si i es bajo, significa que el problema se divide en un número mayor de subproblemas más pequeños, lo que implica que la solución divide y vencerás es más eficiente. Si i es alto, significa que el problema se divide en un número menor de subproblemas más grandes, lo que puede hacer que la solución divide y vencerás sea menos eficiente.  
  
En general, queremos que k y c sean bajos y que i sea alto para obtener una solución divide y vencerás más eficiente.  
  
b) Para determinar cuál de las dos variantes de multiplicación de enteros grandes es más eficiente en términos de la solución divide y vencerás, debemos analizar cómo se relacionan con los parámetros k, c e i.  
  
En la Variante 1, la expresión tiene tres términos: aci^n, (ad+bc)i^(n/2) y bd. El primer término tiene un factor k igual a ac, el segundo término tiene un factor k igual a ad+bc, y el tercer término tiene un factor k igual a bd. Si comparamos estos factores k con c^i, podemos ver que el primer término tiene un factor k menor que c^i, el segundo término tiene un factor k igual a c^i, y el tercer término tiene un factor k mayor que c^i. Por lo tanto, podemos concluir que la complejidad computacional de la Variante 1 es O(n^i log n).  
  
En la Variante 2, la expresión tiene tres términos: aci^n, ((a-b)(d-c)+ac+bd)i^(n/2) y bd. El primer término tiene un factor k igual a ac, el segundo término tiene un factor k igual a (a-b)(d-c)+ac+bd, y el tercer término tiene un factor k igual a bd. Si comparamos estos factores k con c^i, podemos ver que el primer término tiene un factor k menor que c^i, el segundo término tiene un factor k mayor que c^i, y el tercer término tiene un factor k mayor que c^i. Por lo tanto, podemos concluir que la complejidad computacional de la Variante 2 es O(n^(log\_c k)).  
  
En términos de la solución divide y vencerás, la Variante 1 es más eficiente que la Variante 2, ya que tiene una complejidad computacional de O(n^i log n), mientras que la Variante 2 tiene una complejidad computacional de O(n^(log\_c k)). Esto se debe a que en la Variante 1, el segundo término tiene un factor k igual a c^i, lo que implica que los subproblemas son comparables en tamaño al problema original. En cambio, en la Variante 2, el segundo término tiene un factor k mayor que c^i, lo que implica que los subproblemas son más grandes en comparación con el problema original.  
  
c) La complejidad computacional de una búsqueda secuencial sin centinela es O(n), donde n es el tamaño del conjunto de datos en el que se realiza la búsqueda. Esto se debe a que en el peor caso, la búsqueda debe recorrer todos los elementos del conjunto de datos para determinar si el elemento buscado está presente o no.  
  
Por otro lado, la complejidad computacional de una búsqueda secuencial con centinela es O(n+1), donde n es el tamaño del conjunto de datos en el que se realiza la búsqueda. Esto se debe a que se agrega un elemento adicional (el centinela) al final del conjunto de datos, lo que permite que la búsqueda se realice sin tener que verificar si se ha alcanzado el final del conjunto de datos en cada iteración.  
  
En comparación con la solución divide y vencerás, tanto la búsqueda secuencial sin centinela como la búsqueda secuencial con centinela tienen una complejidad computacional más alta. La solución divide y vencerás tiene una complejidad computacional más eficiente, ya que puede reducir el tamaño del problema original dividiéndolo en subproblemas más pequeños, lo que permite una resolución más rápida. En cambio, la búsqueda secuencial requiere recorrer todos los elementos del conjunto de datos en el peor caso, lo que puede llevar más tiempo.>