Examen #3

Matemáticas
Discretas

Acosta Imandt Daniel

316002228

06/Noviembre/2021

Problema 1

(2.5 pts) Los números de Tribonacci T_n están definidos recursivamente como sigue. Se tiene que $T_0 = 0, T_1 = 0 y T_2 = 1$. Además, para cada entero $n \geq 0$ se tiene que $T_{n+3} = T_n + T_{n+1} + T_{n+2}$. Diseña un algoritmo de programación dinámica que calcule los n´umeros de Tribonacci y explica cómo lo usarías para determinar cuáles de los Tribonacci con n entre 1 y 10000 tienen la mayor cantidad de dígitos iguales a 1.

Solución

Lo haremos por memoización.
Idea del algoritmo Primero vamos a crear una lista $T = [0, 0, 1]$, después hacemos una función mem_{Tri} en la cual pedimos el número que se quiere calcular y la lista T , vemos si la longitud de la lista es mayor al número buscado de ser así regresamos $T[n]$, de otra forma vamos a utilizar recursión, en donde creamos una nueva variable new_T la cual es la suma de $mem_{tri}(n-1, T)$,$mem_{tri}(n-2, T)$,$mem_{tri}(n-3, T)$ y el valor obtenido lo adjuntamos a la lista T y regresamos el valor de la suma pasada.

Algorithm 1 Algoritmo para calcular el n-ésimo número de Tribonacci

Require: T=[0,0,1], el n-ésimo número que se quiere calcular

1: **if** $len(T) > n$ **then**
2: **return** $T[n]$
3: **else**
4: $new_{tri} = algoritmo(n-1, T) + algoritmo(n-2, T) + algoritmo(n-3, T)$
5:
6: $T.append(new_tri)$
7: **return** algoritmo(n-1,T)+algoritmo(n-2,T)+algoritmo(n-3,T)
8:
9: **end if**

Ensure: El número de Tribonacci para el n dado

Ahora voy a explicar como usaría este algoritmo para determinar cuáles de los números de Tribonacci con n entre 1 y 10000 tienen la mayor cantidad de dígitos iguales a 1.

1. Hago un ciclo for de n a 10000 en donde llamo la función mem_{tri}
2. imprimo $mem_{tri}(i, T)$ y además convierto ese número a un str, donde uso count(1) y me muestra la cantidad de veces que aparece el dígito 1 .
3. Veo el número de Tribonacci el cual tiene el dígito 1 más veces con max(lista).

Problema 2

(2.5 pts) Diseña un algoritmo por backtrack que encuentre el conjunto independiente más grande de una gráfica con n vértices . Tu algoritmo deberá de descartar un vector candidato de vértices una vez que cuente con dos vértices adyacentes, o bien elegir los candidatos para nunca pasar por un vector así. Demuestra que tu algoritmo es correcto y estudia en cuánto tiempo y espacio asintótico corre.

Solución

Primero veremos la definición de conjunto independiente, es aquel que ninguna pareja de vértices en el conjunto es adyacente.
El método de algoritmo por backtrack construye un vector mediante la adición o la eliminación e la parte final de elementos de una lista C de candidatos.

Idea del algoritmo: Creamos una lista que tenga todas las opciones posibles de un conjunto independiente dado la cantidad de vértices.
Para cada elemento de esta lista vemos todos los vértices del vector y vemos si tienen vértices adyacentes, de ser ese el caso pasamos al siguiente valor de la lista,y hacemos lo mismo para todos los elementos de la lista, en caso de que todos los elementos del vector no tengan vértices adyacentes, este vector se va a adjuntar a otra lista C.
Por ultimo contamos la cantidad del elementos que tiene cada vector de la lista C y regresamos el que tenga más.
Ahora vemos por que este algoritmo es eficiente.

Algorithm 2 Encontrar el conjunto independiente de vértices más grande de una gráfica

Require: Gráfica G con una lista de sus vértices V y otra de sus enlaces E

Creamos una lista A con todos los posibles escenarios de conjuntos independientes.
for each $elemento \in \mathcal{A}$ **do**
    Checamos si hay dos vértices que tengan un enlace
    Creamos una lista vacía C
    if elemento tiene enlace **then**
return A
    else
        adjuntamos el elemento s de A a C
return A

        Encontramos el elemento b de C que tenga la mayor cantidad de elementos

Ensure: b

Notamos que primero tenemos toda la lista de subconjuntos posibles de vértices de la gráfica por lo que no dejamos a nadie atrás.Ahora en el ciclo for checamos si existen elementos del subconjunto que comparten algún enlace, en caso de que si se tenga no se hace nada, en caso contrario adjuntamos este subconjunto a una lista de posibles candidatos, una vez que el ciclo haya pasado por todos los subconjuntos posibles nos fijamos en la lista de los posibles candidatos, en donde contamos los elementos de cada subconjunto y por ultimo regresamos el subconjunto que tenga más elementos.Por lo que parece ser que no hay falla en el algoritmo.
Por ultimo nos fijamos en cuanto tiempo y espacio asintótico corre.
Para la complejidad de tiempo notamos que primero comparamos todos los elementos de cada caso posible con la lista de enlaces, y por ultimo comparamos todos los elementos de posibles candidatos, en el peor de los casos, estas dos listas tienen la misma longitud de $2^n - 1$ elementos entonces en la lista de

posibles candidatos se hacen 2^n comparaciones para encontrar el elemento con más vértices y en la primera lista son muchas más comparaciones por que tenemos que comparar todos los vértices entre ellos, en el peor de los casos de estos tiene$n$ elementos por lo que hay que hacer n^2 comparaciones.Entonces recapitulando el ciclo for tiene complejidad de $O(n^2)$ y al checar los posibles candidatos se hacen $O(2^n)$ comparaciones, pero conforme n tiende a infinito el termino de 2^n crece más rápido que el de n^2, por lo que el programa total tiene una complejidad de $O(2^n)$ en tiempo.
Para el espacio notamos que primero generamos una lista de todos los subconjuntos de la gráfica con longitud k y otra de los posibles candidatos,en el peor de los casos todos los posibles candidatos son todos los subconjuntos, por lo que tenemos dos listas de k elementos y estos elementos son listas, por lo que tenemos dos listas de listas, ahora recordamos que la cantidad de subconjuntos de un conjunto de n elementos es $2^n - 1$,Por lo que en espacio tiene una complejidad de $O(2 * 2^n) = O(2^{n+1})$ lo cual no es muy bueno para grandes gráficas.

Problema 3

(2.5 pts) Se tiene un cubo. Un acomodo bueno consiste en poner sobre sus vértices 8 números distintos del 1 al 10. Un acomodo es muy bueno si la suma de los n´umeros en cada cara del cubo es la misma para todas las caras.

- Mediante un algoritmo básico de combinatoria, encuentra la cantidad exacta de acomodos buenos que existen.
- Con base en la respuesta anterior, estima aproximadamente cuántos minutos se tardar´a una computadora de escritorio actual en ejecutar un algoritmo para contar los algoritmos muy buenos que proceda por fuerza bruta verificando todos los acomodos buenos
- Escribe un pseudocódigo del algoritmo de fuerza bruta anterior. Justifica su correctitud.
- Si en vez de ser los números del 1 al 10 tuviéramos los números del 1 al 30, ¿aproximadamente cuánto tomaría la solución por fuerza bruta? ¿qué podrías hacer para que la exploración fuera factible?

Solución

a)
Notamos que tomamos 8 números de 10, entonces hacemos una permutación para ver todos los casos posibles, los cuales son 1814400.
Pero también queremos ver que la suma de los vértices de todas las caras sea la misma.Entonces haremos lo siguiente ya que tenemos nuestra lista de todas las permutaciones posibles , generamos un contador que empieza en cero, y empezamos un ciclo for que va del primer elemento de la lista al ultimo,en donde si se llega al caso en el que la suma de los vértices de todas las caras son iguales entonces al contador se le suma 1, por ultimo regresamos el contador final.
Al implementar este algoritmo nos damos cuenta que existen 1814400 acomodos buenos y 1824 muy buenos
b)
A pesar de que se esta usando fuerza bruta en este algoritmo,el algoritmo corre bastante rápido, en el caso del que implemente se tardó 2.2206 segundos y no solo calcule los muy buenos, sino también los buenos.
También hay que tomar en cuenta que use la función ya pre-hecha de python de permutaciones, si también hubiera creado una función por mi cuenta se hubiera tardado mucho más.Por lo que para este caso es rápido. Por lo que para este caso es rápido.

c)
A continuación escribo el pseudocódigo de fuerza bruta anterior.

Algorithm 3 Algoritmo para encontrar todos los cubos con acomodo muy bueno

Require: El número de vértices y el total de números que se pueden usar

Generamos todas las permutaciones posibles,dado el número de vértices y los números a elegir.
Convertimos estas permutaciones en una lista de listas.

$count \leftarrow 0$
$cb \leftarrow 0$
for each i in len(lista) **do**:

    $cb+ = 1$
    if a[i][0]+a[i][1]+a[i][2]+a[i][3]==a[i][2]+a[i][3]+a[i][4]+a[i][5]
==a[i][2]+a[i][3]+a[i][4]+a[i][5]==a[i][4]+a[i][5]+a[i][6]+a[i][7]==
a[i][2]+a[i][0]+a[i][4]+a[i][6]==a[i][5]+a[i][7]+a[i][3]+a[i][1]==
a[i][6]+a[i][7]+a[i][0]+a[i][1] **then**

        $count+ = 1$

Ensure: El número de acomodos muy buenos

d)
Si ahora en vez de ser 10 números son 30 la cosa se pone mucho más feas ya que la cantidad de permutaciones se hace increíblemente más grande, tratare de correrlo y ver cuanto tarda.(Actualización) se uso toda la memoria RAM y no se pudo :(.
Pero haremos unos simples cálculos vamos que se tardo aproximadamente $2{,}22$ segundos para 10 ahora vemos que para 30 existen $30 * 29 * 28 * 27 * 26 * 25 * 24 * 23 = 235989936000$ permutaciones posibles. Entonces si para 10 fueron 2.22 segundos por regla de tres vemos que para 30 se tarda(si mi computadora lo aguantará) $2{,}861 \times 10^5$ segundos que es aproximadamente 3.312 días Por lo que vemos que el algoritmo es eficiente para un número relativamente pequeño de números,pero para valores más grandes no lo es, entonces hay que encontrar mejores formas de implementar el código en vez de fuerza bruta.

Problema 4

(2.5 pts) Aplica el teorema maestro a las siguientes recursiones para cerrarlas asintóticamente, o bien explica por qué no puede ser usado.

- $T(n) = 6T(n/2) + \sqrt{n}$
- $T(n) = 7T(n/2) + 4^n$
- $T(n) = 5T(n/5) + (n/log(n))$

Solución

Primero vamos a recordar el teorema maestro.
Teorema maestro Supongamos que tenemos una función que satisface la siguiente ecuación recursiva:

$T(n) = aT(n/b) + f(n)$
En donde $a \geq 1$ y $b > 1$ son constantes y $f(n)$ es una función positiva. Definamos $d = log_b\, a = loga/logb$ al cual le llamaremos el exponente crítico. Entonces, tenemos los siguientes tres casos:

1. si $f(n) = O(n^{d-\epsilon})$ para alguna constante $\epsilon > 0$ entonces $T(n) = \Theta(n^d)$
2. si $f(n) = \Theta(n^d log^k)$ para alguna $k \geq 0$ entonces $T(n) = \Theta(n^d log^{k+1} n)$
3. Si $f(n) = \Omega(n^{d+\epsilon})$ para alguna constante $\epsilon > 0$ y además $f(n)$ satisface la condición de regularidad $af(n/b) < cf(n)$ para alguna constante $c < 1$ y n suficientemente grande , entonces $T(n) = \Theta(f(n))$

- $T(n) = 6T(n/2) + \sqrt{n}$
  En efecto $a = 6 \geq 1$, $b = 2 > 1$ y $f(n) = \sqrt{n}$ por lo que las hipótesis iniciales se cumplen , aquí el exponente crítico es $d = log6/log2 \approx 2{,}5849$.Notamos que $f(n) = \sqrt{n} = O(n^{log6/log2-2{,}0589})$

  $$\underline{T(n) = \Theta(n^{log6/log2})}$$

- $T(n) = 7T(n/2) + 4^n$
  En efecto $a = 7 \geq 1$, $b = 2 > 1$ y $f(n) = 4^n$ por lo que las hipótesis iniciales se cumplen , aquí el exponente crítico es $d = log7/log2 \approx 2{,}807$.
  Ahora queremos que $d + \epsilon = 3$, ahora vemos que para una n_0 suficientemente grande,en este caso $n_0 = 3$ y una $c = 1 - \epsilon_1$ con $\epsilon_1 > 0$ y muy pequeña,se cumple que para toda $n \geq n_0$ tenemos que $f(n) = 4^n \geq n^3 = n^{d+\epsilon}$, ahora veremos que esto se cumple para toda $n > 3$ lo hacemos por inducción, ya tenemos nuestro paso inductivo en donde $4^3 > 3^3$.
  Ahora para nuestra hipótesis inductiva suponemos que se cumple para una n que $4^n \geq n^3$.
  Por ultimo vemos que para $n + 1$ tenemos lo siguiente:

  $$4^{n+1} = 4 * 4^n \geq 4n^3 = n^3 + n^3 + n^3 + n^3 \geq n^3 + 3n^2 + 3n + 1 = (n+1)^3$$

  Por lo tanto vemos que la desigualdad $4^n \geq n^3$ se cumple para toda $n \geq 3$.
  Con esto podemos decir que $f(n)$ cumple la condición de regularidad, además notamos lo siguiente que se cumple para $n \geq 3$.

  $$af(n/2) = 7 * 4^{n/2} = 7 * 2^n < c * 2^n \approx 2^n$$

  Por lo tanto con todo esto y por el teorema maestro podemos afirmar que

  $$\underline{T(n) = \Theta(4^n)}$$

- $T(n) = 5T(n/5) + (n/log(n))$
  En efecto $a = 5 \geq 1$, $b = 5 > 1$ y $f(n) = (n/log(n))$ por lo que las hipótesis iniciales se cumplen , aquí el exponente crítico es $d = log5/log5 = 1$.
  Primero creo que esto no jala ya que $f(n)$ no es polinomial, pero no estoy seguro así que checaré cada uno de los casos.
  Primero tratemos de ver si se puede por el caso 1.Aquí vemos que que para algún $\epsilon > 0$ tiene que existir un c y un n_0 tal que $f(n) = \frac{n}{log(n) \leq cn^{d-\epsilon}}$ por lo que llegamos a que:

  $$\frac{\frac{n}{log(n)}}{n^{d-\epsilon}} = \frac{n}{n^{d-\epsilon} log(n)} = \frac{n^\epsilon}{log(n)} \leq c$$

  Y conforme n tiende a infinito llegamos a que esta función se va a infinito, por lo que no esta acotado por una c, por lo que el caso 1 falla.

Notamos que $f(n) = (n/log(n))$ Aquí vemos que el caso 2 no se puede usar debido a que $\frac{1}{log(n)} = log(n)^{-1}$ pero por hipótesis $k \geq 1$ y aquí no pasa eso.
Por ultimo checamos el tercer caso, el cual tampoco se cumple ya que para cualquier $\epsilon > 0$ y alguna $1 > c > 0$ llegamos a que

$$cn^{d+\epsilon} > n > \frac{n}{log(n)}$$

Por lo que no se da el caso en el que $f(n) = \Omega(n^{d+\epsilon})$.
Por lo tanto este problema no tiene solución con el Teorema maestro

Problema 5 extra

(+2 pts extra) Implementa algoritmos para los problemas 1 y 3 en Python. Tus algoritmos deben de dar la respuesta correcta y correr cada uno de ellos en menos de un minuto

Código

Aquí esta el código para los problemas 1 y 3. NB
De todas formas dejo el enlace del código en los comentarios del examen.
Como se puede ver los códigos corren en mucho menos de un minuto.