Dinamik Programlama Nedir?

Çalışma Kağıdı

Dinamik Programlama, bir problemi, aynı alt problemlerin tekrar tekrar çözülmesini önlemek için çözümlerini saklayarak (memoization veya tabulation yoluyla) daha küçük alt problemlere ayırıp çözen bir yöntemdir.

Sorular

1. Aşağıdakilerden hangisi Dinamik Programlama için temel bir gereklilik DEĞİLDİR?

A) Optimal Alt Yapı
B) Tekrar Eden Alt Problemler
C) Rastgele Sayı Üretimi
D) Çözümleri Saklama (Memoization/Tabulation)

2. Fibonacci dizisinin hesaplanmasında Dinamik Programlama kullanmanın ana avantajı nedir?

A) Hesaplama hızını düşürür.
B) Tekrar eden hesaplamaları önleyerek verimliliği artırır.
C) Daha fazla bellek kullanır.
D) Sadece büyük sayılar için çalışır.

3. Bir problemi Dinamik Programlama ile çözmek için öncelikle ne yapılmalıdır?

A) Problemi rastgele alt problemlere bölmek.
B) Problemin tekrar eden alt problemlere sahip olup olmadığını ve optimal alt yapı özelliğini kontrol etmek.
C) Problemi sadece özyinelemeli olarak çözmek.
D) Problemi en büyük alt problemden başlayarak çözmek.

4. Fibonacci Sayıları Hesabı

5. En Kısa Yol Problemi (Bellman-Ford Algoritması)

6. 0/1 Sırt Çantası Problemi

7. Tanımla: Dinamik Programlama'nın temel amacı nedir?

8. Tanımla: Dinamik Programlama'da 'memoization' ne anlama gelir?

9. Tanımla: 'Tabulation' yöntemi nedir?

Cevap Anahtarı

1. C) Rastgele Sayı Üretimi — Rastgele sayı üretimi dinamik programlamanın temel bir gerekliliği değildir. Optimal alt yapı ve tekrar eden alt problemler, DP'nin uygulanabilmesi için kritik öneme sahiptir.

2. B) Tekrar eden hesaplamaları önleyerek verimliliği artırır. — Dinamik programlama, Fibonacci gibi tekrar eden alt problemlere sahip dizilerde, daha önce hesaplanmış değerleri saklayarak gereksiz tekrarlı hesaplamaları önler ve bu da algoritmanın verimliliğini önemli ölçüde artırır.

3. B) Problemin tekrar eden alt problemlere sahip olup olmadığını ve optimal alt yapı özelliğini kontrol etmek. — Dinamik programlamanın uygulanabilirliği için problemin tekrar eden alt problemlere sahip olması ve optimal alt yapı özelliğini taşıması şarttır. Bu kontrol yapıldıktan sonra çözüm yöntemleri (memoization veya tabulation) belirlenir.

4. 1. Temel durumları tanımla: F(0)=0, F(1)=1. 2. Tekrar eden alt problemler: F(n) = F(n-1) + F(n-2). 3. Çözümleri sakla: Bir dizi veya harita kullanarak hesaplanan Fibonacci değerlerini depola. 4. Özyinelemeli veya döngüsel olarak hesapla: Eğer F(k) daha önce hesaplandıysa saklanan değeri kullan, aksi halde hesapla ve sakla.

5. 1. Başlangıç düğümünün uzaklığını 0, diğer tüm düğümlerin uzaklığını sonsuz olarak ayarla. 2. Kenar sayısının bir eksiği kadar tekrarla: Her adımda, tüm kenarlar için 'gevşetme' işlemi yap. 3. Gevşetme: Eğer u'dan v'ye bir kenar varsa ve dist(u) + ağırlık(u,v) < dist(v) ise, dist(v) = dist(u) + ağırlık(u,v) olarak güncelle. 4. Negatif döngü kontrolü: Ek bir adımda tekrar gevşetme yaparak negatif döngü olup olmadığını kontrol et.

6. 1. Durumu tanımla: dp[i][w], ilk i öğe kullanılarak maksimum w ağırlığındaki sırt çantasına sığabilecek maksimum değeri temsil eder. 2. Temel durum: dp[0][w] = 0 ve dp[i][0] = 0. 3. Özyineleme ilişkisi: Eğer i'inci öğenin ağırlığı w'dan küçükse, dp[i][w] = max(dp[i-1][w], değer[i] + dp[i-1][w - ağırlık[i]]). Aksi halde, dp[i][w] = dp[i-1][w]. 4. Sonuç: dp[n][W] (n: toplam öğe sayısı, W: sırt çantası kapasitesi).

7. Karmaşık problemleri daha küçük, tekrar eden alt problemlere ayırarak ve bu alt problemlerin çözümlerini saklayarak verimli çözümler üretmektir.

8. Alt problemlerin hesaplanan çözümlerini bir veri yapısında (örneğin dizi veya harita) saklayarak, aynı alt problem tekrar çözülmek istendiğinde saklanan değeri kullanma tekniğidir.

9. Problemi en küçük alt problemlerden başlayarak yukarı doğru (bottom-up) hesaplama ve sonuçları bir tabloya (genellikle dizi) doldurma tekniğidir.