Algorithm Analysis And Design

With Infinity Teams

Eng: Abdulrahman Hamdi

INFINITY TEAM
Since 2020

Table of contents

Eng: Abdulrahman Hamdi

Table of contents

Eng: Abdulrahman Hamdi

07

Decrease-and-Conquer

Eng: Abdulrahman Hamdi

Decrease-and-Conquer Çerçevesi

• Insertion Sort

• Topological Sorting

• Algorithms for Generating Combinatorial Objects
- Generating Permutations
- Generating Subsets

• Decrease-by-a-Constant-Factor Algorithms
- Fake-Coin Problem
- Russian Peasant Multiplication
- Josephus Problem

• Variable-Size-Decrease Algorithms
- Computing a Median and the Selection Problem
- Interpolation Search
- Searching and Insertion in a Binary Search Tree
- The Game of Nim

Eng: Abdulrahman Hamdi

Decrease-and-Conquer Algoritmaları

Algoritma dizaynında bir diğer önemli teknik Decrease and Conquer (Azalta ve Yönet) tekniği, bir problemi daha küçük bir hale indirgemeyi (decrease), bu küçük versiyonu çözmeyi ve elde edilen çözümü kullanarak orijinal problemi tamamlamayı esas alır. Buradaki temel nokta, küçük boyutlu problemin çözümünü büyük problemle ilişkilendirebilmektir.Bu teknik, yukarıdan aşağıya (top-down) veya aşağıdan yukarıya (bottom-up) yaklaşımla uygulanabilir. Top-down yaklaşımı genellikle özyinelemeli (recursive) bir yapıya yol açarken, bottom-up yaklaşımı genellikle yinelemeli (iterative) olarak uygulanır ve problemin en küçük haliyle başlar. Bu yönteme bazen artımlı (incremental) yaklaşım da denir.

Decrease and Conquer tekniğinin üç temel varyasyonu vardır:

1- Sabit bir miktar azaltma – Problemin boyutu belirli bir miktar azaltılır.

- Eklemeli Sıralama (Insertion Sort)
- Topolojik Sıralama (Topological Sort)
- Permütasyon, alt küme oluşturma algoritmaları

2- Sabit bir faktörle azaltma – Problemin boyutu belirli bir oranda küçültülür.

- İkili Arama (Binary Search)

3- Değişken boyutlu azaltma – Problemin boyutu her adımda farklı oranlarda azaltılır.

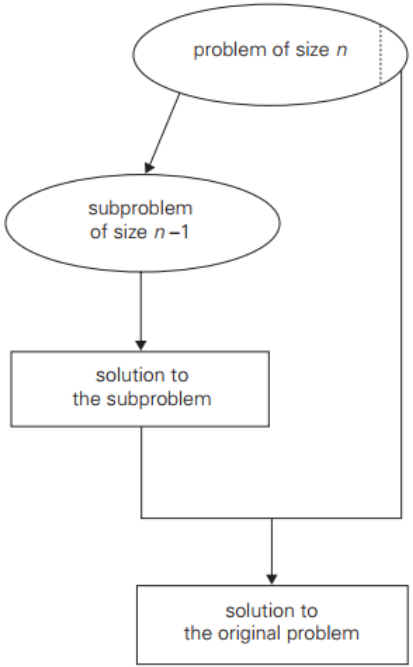
- Öklid algoritması

Eng: Abdulrahman Hamdi

Decrease-and-Conquer Algoritmaları

• **Sabit bir miktar azaltma**
• an üstel problemi
– $a^n = a^{n-1}.a$ ise
– F(n) = a^n problemi
özyinelemeli olarak çözülebilir.

$$f(n) = \begin{cases} f(n-1) \cdot a & \text{if } n > 0, \\ 1 & \text{if } n = 0, \end{cases}$$

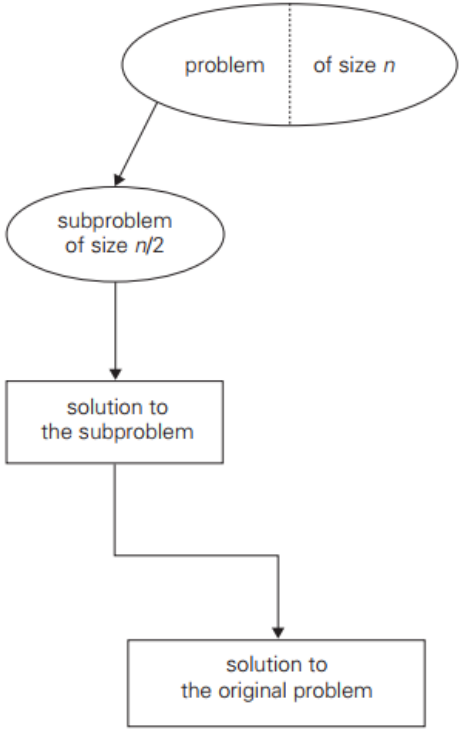


Decrease-and-Conquer Algoritmaları

- **Sabit bir faktörle azaltma**
- an üstel problemi

– $a^n = \left(a^{\frac{n}{2}}\right)^2$ ise.

$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even and positive,} \\ (a^{(n-1)/2})^2 \cdot a & \text{if } n \text{ is odd,} \\ 1 & \text{if } n = 0. \end{cases}$$



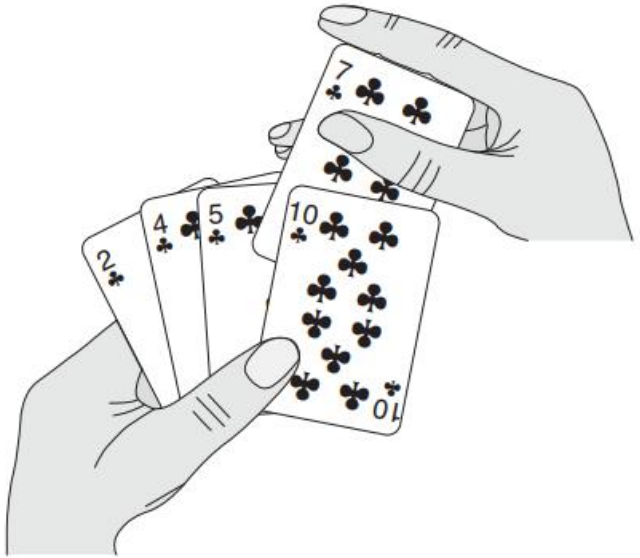
Decrease-and-Conquer Algoritmaları

• **Değişken Oranla Azaltma**
– Öklid'in EBOB algoritması
gcd(m, n) = gcd(n, m mod n).
Burada da orijinal problem daha küçük boyutlu benzer bir problemi indirgenmektedir bu haliyle Öklid yöntemi ile çözülen aşağıdaki gcd algoritması Decrease and conquer tipinde bir algoritmadır.

```
Algorithm EuclidGCD(m, n)
    While n ≠ 0 Do
        r ← m mod n
        m ← n
        n ← r
    return m
```

Decrease-and-Conquer Algoritmaları

• **Eklemeli Sıralama (Insertion Sort)**
Insertion sort, her bir öğeyi sırasız bir listenin sıralı kısmına doğru şekilde yerleştirerek çalışan basit bir sıralama algoritmasıdır. Bu, elinizdeki iskambil kartlarını sıralamaya benzer. Kartları iki gruba ayırırsınız: sıralı kartlar ve sırasız kartlar. Sonra, sırasız gruptan bir kart alır ve onu sıralı grupta doğru yere yerleştirirsiniz.İlk elemanın zaten sıralı olduğu varsayılarak, dizinin ikinci elemanından başlanır. İkinci eleman, ilk elemanla karşılaştırılır; eğer ikinci eleman daha küçükse, kartlar yer değiştirir. Üçüncü elemanla devam edilir, ilk iki elemanla karşılaştırılır ve doğru pozisyona yerleştirilir. Bu işlem, tüm dizi sıralanana kadar tekrarlanır.

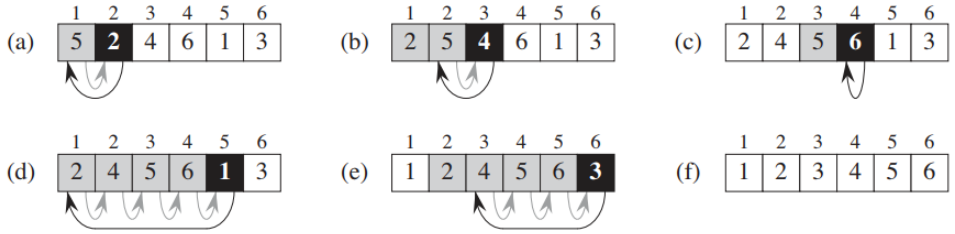


Eng: Abdulrahman Hamdi

Decrease-and-Conquer Algoritmaları

• **Eklemeli Sıralama (Insertion Sort)**

Decrease and Conquer tekniği ile üretilen en popüler algoritmalardan biri, aldığı bir diziyi küçükten büyüğe sıralayan Insertion Sort algoritmasıdır. Bu algoritma, dizinin yalnızca ilk elemanını düşünerek boyutunu azaltır. Daha sonra, birer birer eleman alarak mevcut sıralamayı bozmadan uygun konuma yerleştirir.

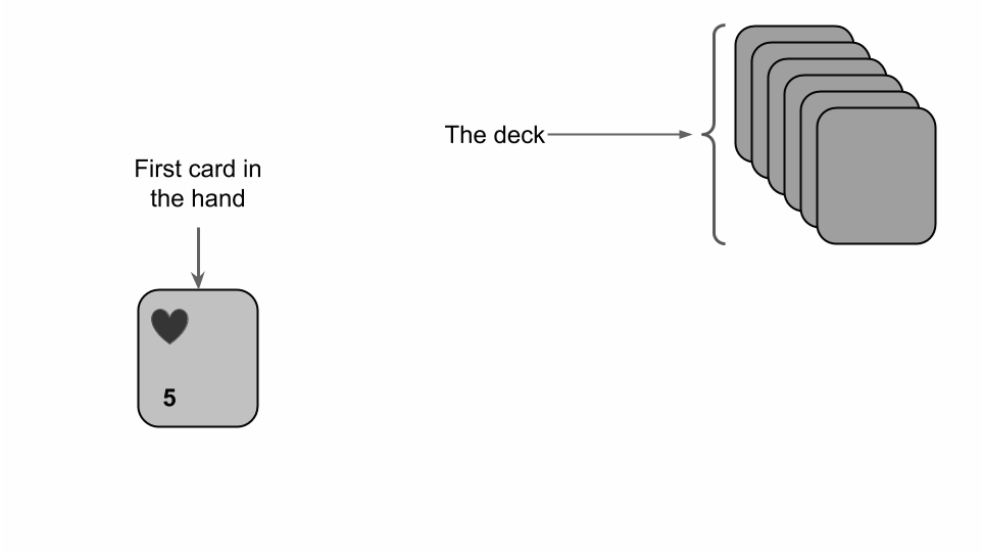


INSERTION-SORT işleminin A = {5, 2, 4, 6, 1, 3} dizisi üzerindeki çalışması. Dizi indisleri dikdörtgenlerin üstünde, dizinin içindeki değerler ise dikdörtgenlerin içinde gösterilmektedir.(a)–(e) for döngüsünün 1–8. satırları boyunca gerçekleşen yinelemeleri gösterir. Her yinelemede, siyah dikdörtgen dizinin A[j] elemanını tutar ve bu eleman, 5. satırdaki testte solundaki gölgelendirilmiş dikdörtgenlerdeki değerlerle karşılaştırılır. Gölgelendirilmiş oklar, 6. satırda dizideki değerlerin bir sağa kaydırıldığını gösterirken, siyah oklar, 8. satırda anahtarın nereye yerleşeceğini gösterir.(f) Son olarak, dizinin sıralanmış hali gösterilmektedir.

Eng: Abdulrahman Hamdi

Decrease-and-Conquer Algoritmaları

- **Eklemeli Sıralama (Insertion Sort)**



6 5 3 1 8 7 2 4

Decrease-and-Conquer Algoritmaları

• **Eklemeli Sıralama (Insertion Sort)**

– Versiyon 1:

```
Algorithm INSERTION-SORT(A)
1.  for j ← 2 to length(A) do
2.      key ← A[j]
3.// Insert A[j] into the sorted sequence
    A[1 . . j-1].
4.      i ← j - 1
5.      while i > 0 and A[i] > key do
6.          A[i + 1] ← A[i]
7.          i ← i - 1
8.      A[i + 1] ← key
```

Döngü İnvariantları ve Insertion Sort'un Doğruluğu:

Insertion Sort algoritmasının nasıl çalıştığını anlamak için döngü invariantlarını kullanırız. A = {5, 2, 4, 6, 1, 3} dizisi üzerinde algoritmanın çalışma adımlarını göstermektedir. Burada j değişkeni, "eldeki mevcut kartı" temsil eder ve her yinelemede uygun konuma yerleştirilir.

Döngü İnvariantı Nedir?

Bir döngü invariantı, her döngü yinelemesi öncesinde doğru olan bir özelliktir. Insertion Sort için döngü invariantı şu şekildedir:

Her for döngüsü yinelemesi başlangıcında, A[1..j-1] alt dizisi sıralıdır ve bu alt dizi, başlangıçta dizinin ilk j-1 elemanını içerir.

Eng: Abdulrahman Hamdi

Decrease-and-Conquer Algoritmaları

• Eklemeli Sıralama (Insertion Sort)

Döngü invariantının algoritmanın doğruluğunu kanıtlamak için üç aşamada incelenmesi gerekir:

- Başlangıç (Initialization):

Döngü başladığında (j = 2), A[1] tek elemanlıdır ve doğal olarak sıralıdır.Bu, döngü invariantının ilk adımda geçerli olduğunu gösterir.

- Koruma (Maintenance):

Döngü her yinelemede, A[j] değerini uygun konuma yerleştirmek için önceki elemanları kaydırarak sıralamayı bozmadan ekleme yapar.Böylece, A[1..j] her adımda sıralı olmaya devam eder.Bir sonraki yineleme başladığında, invariant korunmuş olur.

- Sonlanma (Termination):

Döngü j > n olduğunda sona erer.Döngü invariantına göre, A[1..n] sıralanmış bir dizi haline gelmiştir.Bu, algoritmanın doğru çalıştığını kanıtlar.Döngü invariantı, algoritmaların doğruluğunu göstermek için güçlü bir yöntemdir ve ilerleyen bölümlerde diğer algoritmalar için de kullanılacaktır.

Eng: Abdulrahman Hamdi

Decrease-and-Conquer Algoritmaları

• Eklemeli Sıralama (Insertion Sort)

Bir algoritmanın çalışma süresi, yürütülen her ifadenin çalışma sürelerinin toplamıdır. ci adımda çalışan ve n kez yürütülen bir ifade, toplam çalışma süresine ci * n kadar katkı sağlar. INSERTION-SORT algoritmasının n elemanlı bir girdi üzerindeki çalışma süresi T(n) hesaplanırken, her ifadenin maliyet (cost) ve çalıştırılma sayısı (times) çarpımları toplanarak elde edilir.

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .$$

Belirli bir boyuttaki girdiler için bile, bir algoritmanın çalışma süresi, verilen girdiye bağlı olarak değişebilir.

Örneğin, INSERTION-SORT algoritmasında en iyi durum (best case), dizi zaten sıralı olduğunda gerçekleşir. Bu durumda, j = 2, 3, ..., n için i başlangıç değeri 1 olduğunda, A[i] ≤ key koşulu satır 5'te sağlanır.Böylece, en iyi durumda algoritmanın calısma süresi su sekilde olur...

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

Bu çalışma süresini, n'in bir doğrusal fonksiyonu (linear function) olarak ifade edebiliriz.Dizi tersten sıralı (azalan sırada) olduğunda, en kötü durum (worst case) ortaya çıkar.Bu durumda, her A[j] elemanı, önceden sıralanmış alt dizi A[1 . . j - 1] içindeki her elemanla karşılaştırılmalıdır.
Bu yüzden, j = 2, 3, ..., n için karşılaştırma sayısı t_1 = j olur.

Eng: Abdulrahman Hamdi

Decrease-and-Conquer Algoritmaları

- **Eklemeli Sıralama (Insertion Sort)**

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

and

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

(Bu toplamların nasıl çözüleceğini gözden geçirmek için Ek A'ya bakınız.)En kötü durumda, INSERTION-SORT algoritmasının çalışma süresinin şu şekilde olduğunu buluruz...

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

En kötü durum çalışma süresini $an^2 + bn + c$ şeklinde, a, b ve c sabitleriyle ifade edebiliriz. Bu nedenle, çalışma süresi n'in ikinci dereceden (quadratic) bir fonksiyonudur.Insertion sort algoritmasında, belirli bir giriş için çalışma süresi sabittir. Ancak ilerleyen bölümlerde, aynı giriş için farklı davranışlar sergileyebilen "rastgeleleştirilmiş" algoritmalar göreceğiz.

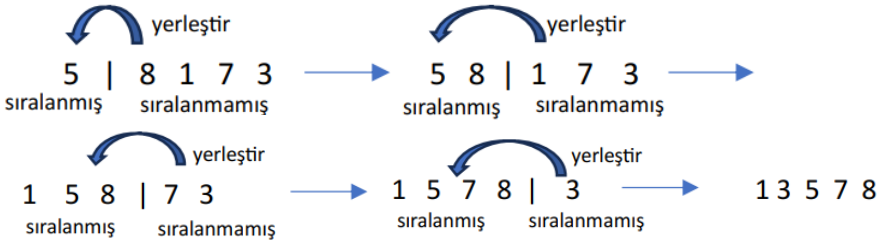
Eng: Abdulrahman Hamdi

Decrease-and-Conquer Algoritmaları

• **Eklemeli Sıralama (Insertion Sort)**

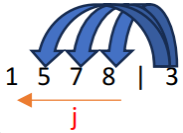
En Kötü Durum ve Ortalama Durum Analizi

Insertion sort analizimizde hem en iyi durumu (giriş dizisi zaten sıralı), hem de en kötü durumu (giriş dizisi ters sıralı) inceledik. Ancak, kitabın geri kalanında genellikle en kötü durum çalışma süresini analiz etmeye odaklanacağız. Yani, **n** boyutundaki herhangi bir giriş için en uzun çalışma süresini bulmaya çalışacağız.



Demek ki 2.pozisyondan (1.indis) başlayarak sol tarafa doğru uygun bir yere yerleştirme yapacağız. Nereye yerleştireceğimizi bulurken çubuğun (|) solundan en başa doğru sağdan sola yerleştireceğiz.

Örnek olarak:



3 ile bu sayıları kıyaslayacağız 3 bu sayılardan küçük iken sola doğru gideceğiz

Eng: Abdulrahman Hamdi

Decrease-and-Conquer Algoritmaları

• Eklemeli Sıralama (Insertion Sort)

En son j=0 olduğunda A[j]<3 olur. O halde 1'in bir sağına yani j+1 pozisyona 3'u yerleştireceğiz. Fakat bu noktada problem j+1 pozisyonda 5 vardır ve eğer biz 3'ü buraya yerleştirirsek 5'in üzerine 3 yazılmış olur ve 5 kaybolur. Bu problemi aşmanın bir yolu A[j+2]' ye 5'i yazmaktır. Yani her A[j]>3 olduğunda A[j+1]'e A[j] yazılarak A[j] çoğaltılır:

1 5 7 8 13

A[3] > 3 olduğu için 1 5 7 8 8 A[2]>3 olduğu için

A[1]>3 olduğu için 1 5 5 7 8

A[0]<3 olduğu için 1 3 5 7 8

- Versiyon 2:

```
InsertionSort(A[0, ..., n-1])
// Girdi : A uzunluğunda bir A dizisi
// Çıktı : Elemanları küçükten büyüye sıralanmış A

    for i = 1 to n-1 // 2. pozisyondan başlıyoruz
    v = A[i] // yerleştirilecek olan eleman
    j = i - 1 // i'nin bir solundaki ilk pozisyon

        while j >= 0 and A[j] > v
        A[j + 1] = A[j] // A[j]'yi bir sağa kaydır
        j = j - 1 // bir sonraki pozisyona geç
        A[j + 1] = v // v'yi doğru pozisyona
    yerleştir
```

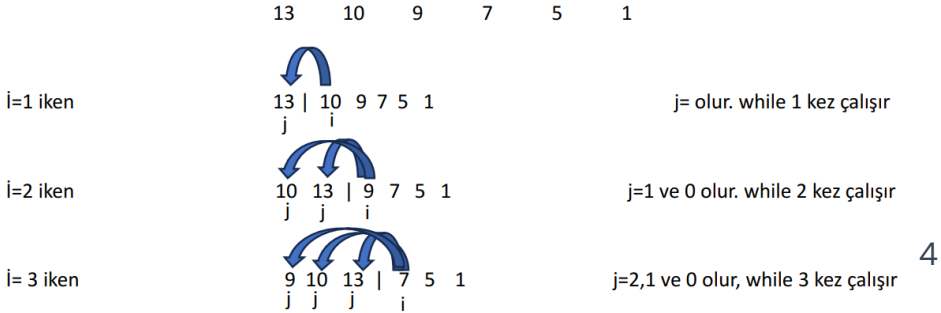
Decrease-and-Conquer Algoritmaları

• **Eklemeli Sıralama (Insertion Sort)**

Simdi insertation sort'un zaman verimliliğini ve bu verimliliğe ait asimptotik analiz yapalım.

1)En iyi durum : en iyi durumda algoritmaya girecek dizi hâlihazırda küçükten büyüğe sıralanmış durumdadır . Her i için A[j]< A[i] olur böylece iç while loopu hiç çalışmaz. Sadece dışarıdaki for loop çalışır. O halde n-1 kez çalışır. Bu durumda $T_{Best}(n) = n - 1 \epsilon \theta(n)$

2) En kotu durum: en kotu durumda dizi büyükten küçüğe doğru sıralanmış halde gelir. Bunu bir örnekle anlatalım.



Bu şekilde görülüyor ki i=4 olduğunda while 4 kez,5 olduğunda 5 kez .... çalışır. Bu durumu genellersek n uzunluğunda bir dizi için iç while loopunun çalışma sayıları toplamda

$$T_{Worst}(n) = 1 + 2 + 3 + \cdots + n - 1 = \frac{n(n-1)}{2} \epsilon \theta(n^2)$$

Decrease-and-Conquer Algoritmaları

• **Eklemeli Sıralama (Insertion Sort)**

- Özet:

89 |	**45**	68	90	29	34	17
45	89 |	**68**	90	29	34	17
45	68	89 |	**90**	29	34	17
45	68	89	90 |	**29**	34	17
29	45	68	89	90 |	**34**	17
29	34	45	68	89	90 |	**17**
17	29	34	45	68	89	90

$$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \Theta(n^2).$$

$$C_{best}(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

- Versiyon 2:

```
InsertionSort(A[0, ..., n-1])
// Girdi : A uzunluğunda bir A dizisi
// Çıktı : Elemanları küçükten büyüye sıralanmış A

    for i = 1 to n-1 // 2. pozisyondan başlıyoruz
    v = A[i] // yerleştirilecek olan eleman
    j = i - 1 // i'nin bir solundaki ilk pozisyon

        while j >= 0 and A[j] > v
        A[j + 1] = A[j] // A[j]'yi bir sağa kaydır
        j = j - 1 // bir sonraki pozisyona geç
        A[j + 1] = v // v'yi doğru pozisyona
    yerleştir
```

Eng: Abdulrahman Hamdi

Decrease-and-Conquer Algoritmaları

• **Topolojik Sıralama (Topological Sorting)**
Bu bölümde, ön koşul kısıtlamalı görevleri içeren çeşitli uygulamaları olan, yönlü graflar için önemli bir problemi ele alacağız. Ancak bu problemi ortaya koymadan önce, yönlü graflar hakkında bazı temel gerçekleri gözden geçirelim. Yönlü bir graf, veya kısaca digraf, tüm kenarları için yön belirtilmiş bir graftır (Şekil a bir örnektir). Bitişiklik matrisi ve bitişiklik listeleri, bir digrafı temsil etmenin hala iki temel yoludur.

Yönlü ve yönsüz grafları temsil etme arasında sadece iki önemli fark vardır: (1) yönlü bir grafın bitişiklik matrisi simetrik olmak zorunda değildir; (2) yönlü bir graftaki bir kenarın, digrafın bitişiklik listelerinde sadece bir (iki değil) karşılık gelen düğümü vardır.

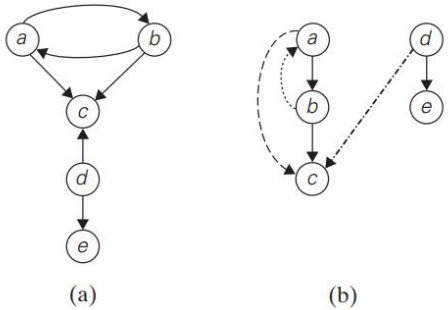


FIGURE (a) Digraph. (b) DFS forest of the digraph for the DFS traversal started at a.

Decrease-and-Conquer Algoritmaları

• **Topolojik Sıralama (Topological Sorting)**
Derinlik öncelikli arama (DFS) ve genişlik öncelikli arama (BFS) yönlü grafları geçmek için temel geçiş algoritmalarıdır, ancak karşılık gelen ormanların yapısı yönsüz graflara göre daha karmaşık olabilir. Bu nedenle, Şekil a'daki basit örnek için bile, derinlik öncelikli arama ormanı (Şekil b), yönlü bir grafın DFS ormanında mümkün olan dört tür kenarın tümünü sergiler: ağaç kenarları (ab, bc, de), tepe noktalarından atalarına geri kenarlar (ba), tepe noktalarından çocukları dışındaki ağaçtaki torunlarına ileri kenarlar (ac) ve yukarıda belirtilen türlerden hiçbiri olmayan çapraz kenarlar (dc).

Yönlü bir grafın DFS ormanındaki bir geri kenarın, bir tepe noktasını ebeveynine bağlayabileceğine dikkat edin. Durum böyle olsun ya da olmasın, bir geri kenarın varlığı, digrafın yönlü bir döngüye sahip olduğunu gösterir.

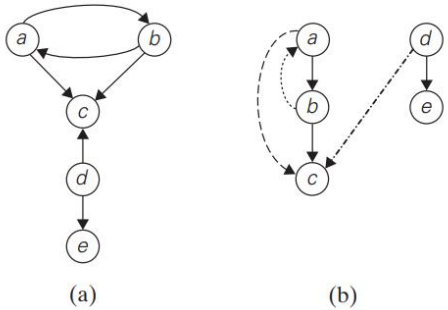


FIGURE (a) Digraph. (b) DFS forest of the digraph for the DFS traversal started at *a*.

Decrease-and-Conquer Algoritmaları

• Topolojik Sıralama (Topological Sorting)
Bir digraftaki yönlü döngü, aynı tepe noktasıyla başlayan ve biten ve her tepe noktasının hemen önceki tepe noktasına, önceki tepe noktasından sonraki tepe noktasına yönlendirilmiş bir kenar ile bağlandığı üç veya daha fazla tepe noktasının bir dizisidir. Örneğin, a, b, a, Şekil a'daki digrafta yönlü bir döngüdür. Tersine, bir digrafın DFS ormanında geri kenarlar yoksa, digraf bir dag'dır yani yönlü asiklik grafın kısaltmasıdır.
Kenar yönleri, yönsüz graflar için ya anlamsız ya da önemsiz olan digraflar hakkında yeni sorulara yol açar. Bu bölümde, bu tür bir soruyu ele alacağız.

Motive edici bir örnek olarak, yarı zamanlı bir öğrencinin bir derece programında alması gereken beş zorunlu dersin {C1, C2, C3, C4, C5} kümesini düşünün.

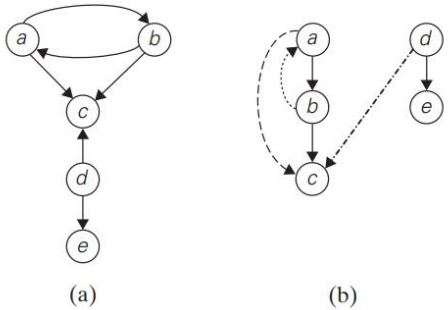


FIGURE (a) Digraph. (b) DFS forest of the digraph for the DFS traversal started at *a*.

Eng: Abdulrahman Hamdi

Decrease-and-Conquer Algoritmaları

• **Topolojik Sıralama (Topological Sorting)**
Dersler, aşağıdaki ders ön koşulları karşılandığı sürece herhangi bir sırayla alınabilir: C1 ve C2'nin ön koşulu yoktur, C3, C1 ve C2 gerektirir, C4, C3 gerektirir ve C5, C3 ve C4 gerektirir. Öğrenci her dönem sadece bir ders alabilir. Öğrenci dersleri hangi sırayla almalıdır? Motive edici bir örnek olarak, yarı zamanlı bir öğrencinin bir derece programında alması gereken beş zorunlu dersin {C1, C2, C3, C4, C5} kümesini düşünün. Dersler, aşağıdaki ders ön koşulları karşılandığı sürece herhangi bir sırayla alınabilir: C1 ve C2'nin ön koşulu yoktur, C3, C1 ve C2 gerektirir, C4, C3 gerektirir ve C5, C3 ve C4 gerektirir.

Öğrenci her dönem sadece bir ders alabilir. Öğrenci dersleri hangi sırayla almalıdır?

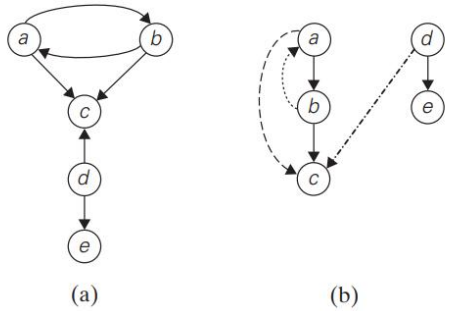


FIGURE (a) Digraph. (b) DFS forest of the digraph for the DFS traversal started at a.

Eng: Abdulrahman Hamdi

Decrease-and-Conquer Algoritmaları

• **Topolojik Sıralama (Topological Sorting)**
Durum, tepe noktalarının dersleri temsil ettiği ve yönlü kenarların ön koşul gereksinimlerini gösterdiği bir digraf ile modellenebilir (Şekil). Bu digraf açısından soru, grafikteki her kenar için, kenarın başladığı tepe noktasının kenarın bittiği tepe noktasından önce listeleneceği şekilde tepe noktalarını listeleyip listeleyemeyeceğimizdir.

Motive edici bir örnek olarak, yarı zamanlı bir öğrencinin bir derece programında alması gereken beş zorunlu dersin {C1, C2, C3, C4, C5} kümesini düşünün. (Bu digrafın tepe noktalarının böyle bir sıralamasını bulabilir misiniz?) Bu probleme topolojik sıralama denir. Bir için sorulabilir.

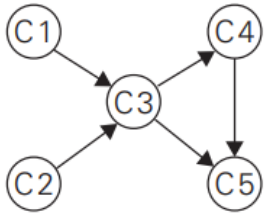


FIGURE Digraph representing the prerequisite structure of five courses.

Decrease-and-Conquer Algoritmaları

• **Topolojik Sıralama (Topological Sorting)**
Keyfi bir digraf için, ancak bir digrafın yönlü bir döngüsü varsa problemin bir çözümü olamayacağını görmek kolaydır. Bu nedenle, topolojik sıralamanın mümkün olması için, söz konusu digrafın bir dag olması gerekir. Bir dag olmak, topolojik sıralamanın mümkün olması için sadece gerekli değil, aynı zamanda yeterlidir; yani, bir digrafın yönlü döngüsü yoksa, topolojik sıralama problemi için bir çözüm vardır. Ayrıca, bir digrafın bir dag olup olmadığını doğrulayan ve eğer öyleyse, topolojik sıralama problemini çözen bir tepe noktası sıralaması üreten iki verimli algoritma vardır.

İlk algoritma, derinlik öncelikli aramanın (DFS) basit bir uygulamasıdır: bir DFS geçişi gerçekleştirin ve tepe noktalarının çıkmaz hale geldiği (yani, geçiş yığınından çıkarıldığı) sırayı not edin. Bu sırayı tersine çevirmek, elbette geçiş sırasında geri bir kenar bulunmamışsa, topolojik sıralama problemine bir çözüm sağlar. Geri bir kenar bulunmuşsa, digraf bir dag değildir ve tepe noktalarının topolojik sıralaması imkansızdır.

Eng: Abdulrahman Hamdi

Decrease-and-Conquer Algoritmaları

• **Topolojik Sıralama (Topological Sorting)**
Algoritma neden çalışır? Bir v tepe noktası bir DFS yığınından çıkarıldığında, u'dan v'ye bir kenarı olan hiçbir u tepe noktası, v'den önce çıkarılan tepe noktaları arasında olamaz. (Aksi takdirde, (u, v) geri bir kenar olurdu.) Bu nedenle, bu tür herhangi bir u tepe noktası, çıkarılan sıra listesinde v'den sonra ve tersine çevrilmiş listede v'den önce listelenecektir.

Şekil a, bu algoritmanın Şekil önce'deki digrafa uygulanmasını göstermektedir. Şekil c'de, digrafın kenarlarını çizdiğimize ve hepsinin problemin ifadesinin gerektirdiği gibi soldan sağa doğru işaret ettiğine dikkat edin. Bu, topolojik sıralama probleminin bir örneğine bir çözümün doğruluğunu görsel olarak kontrol etmenin uygun bir yoludur.

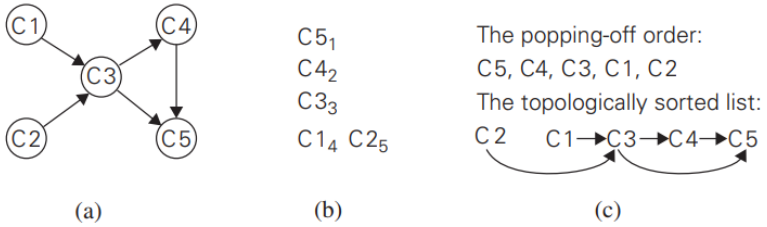


FIGURE (a) Digraph for which the topological sorting problem needs to be solved. (b) DFS traversal stack with the subscript numbers indicating the popping-off order. (c) Solution to the problem.

Decrease-and-Conquer Algoritmaları

• **Topolojik Sıralama (Topological Sorting)**

İkinci algoritma, azalt-ve-fethet tekniğinin (bir azaltma) doğrudan bir uygulamasına dayanmaktadır: kalan bir digrafta tekrar tekrar, gelen kenarı olmayan bir tepe noktası olan bir kaynağı tanımlayın ve ondan çıkan tüm kenarlarla birlikte silin. (Birden fazla kaynak varsa, bağı keyfi olarak kırın. Hiçbiri yoksa, problem çözülemediği için durun—bu bölümün alıştırmalarındaki Problem 6a'ya bakın.) Tepe noktalarının silindiği sıra, topolojik sıralama problemine bir çözüm sağlar. Bu algoritmanın beş dersi temsil eden aynı digrafa uygulanması Şekil'de verilmiştir. aynak kaldırma algoritmasıyla elde edilen çözümün, DFS tabanlı algoritmayla elde edilen çözümden farklı olduğuna dikkat edin. Elbette her ikisi de doğrudur; topolojik sıralama probleminin birkaç alternatif çözümü olabilir. Kullandığımız örneğin küçük boyutu, topolojik sıralama problemi hakkında yanlış bir izlenim yaratabilir.

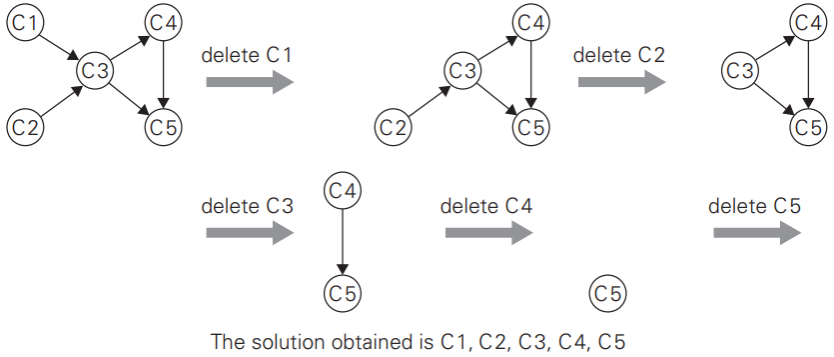


FIGURE Illustration of the source-removal algorithm for the topological sorting problem. On each iteration, a vertex with no incoming edges is deleted from the digraph.

Eng: Abdulrahman Hamdi

Decrease-and-Conquer Algoritmaları

• **Topolojik Sıralama (Topological Sorting)**
Ancak, çok sayıda birbiriyle ilişkili ve bilinen ön koşulları olan görevleri içeren büyük bir projeyi—örneğin inşaat, araştırma veya yazılım geliştirmede—hayal edin. Böyle bir durumda yapılması gereken ilk şey, verilen ön koşullar kümesinin çelişkili olmadığından emin olmaktır. Bunu yapmanın uygun yolu, projenin digrafı için topolojik sıralama problemini çözmektir. Ancak o zaman, örneğin projenin toplam tamamlanma süresini en aza indirmek için görevleri planlamayı düşünmeye başlayabilirsiniz.

Bu, elbette, yöneylem araştırması üzerine genel kitaplarda veya CPM (Kritik Yol Yöntemi) ve PERT (Program Değerlendirme ve Gözden Geçirme Tekniği) metodolojileri üzerine özel olanlarda bulabileceğiniz diğer algoritmaları gerektirecektir. Bilgisayar biliminde topolojik sıralamanın uygulamalarına gelince, bunlar program derlemesinde komut çizelgeleme, elektronik tablo formüllerinde hücre değerlendirme sıralaması ve bağlayıcılarda sembol bağımlılıklarını çözmeyi içerir.

Eng: Abdulrahman Hamdi

Decrease-and-Conquer Algoritmaları

• Kombinatoryal Nesneler Üretmek İçin Algoritmalar

Bu bölümde, kombinatoryal nesneleri üretmek için algoritmaları tartışma sözümüzü tutuyoruz. Kombinatoryal nesnelerin en önemli türleri, verilen bir kümenin permütasyonları, kombinasyonları ve alt kümeleridir. Bunlar tipik olarak farklı seçeneklerin dikkate alınmasını gerektiren problemlerde ortaya çıkar. Kapsamlı aramayı tartıştığımız 3. Bölümde bunlarla zaten karşılaşmıştık. Kombinatoryal nesneler, kombinatorik adı verilen ayrık matematiğin bir dalında incelenir.

Matematikçiler, elbette, öncelikle farklı sayma formülleriyle ilgilenirler; bu tür formüller için minnettar olmalıyız çünkü bize kaç öğenin üretilmesi gerektiğini söylerler. Özellikle, kombinatoryal nesnelerin sayısının tipik olarak problem boyutunun bir fonksiyonu olarak üstel veya daha da hızlı büyüdüğü konusunda bizi uyarırlar. Ancak buradaki temel ilgi alanımız, onları saymaktan ziyade kombinatoryal nesneleri üretmek için algoritmalardır.

Eng: Abdulrahman Hamdi

Decrease-and-Conquer Algoritmaları

• Permütasyon Üretme

Permütasyonlarla başlıyoruz. Basitlik adına, permütasyonları alınması gereken temel kümenin sadece 1'den n'ye kadar olan tamsayılar kümesi olduğunu varsayıyoruz; daha genel olarak, bunlar n elemanlı bir kümedeki a_1, ..., an elemanlarının indeksleri olarak yorumlanabilir. 1,..., n'nin tüm n! permütasyonunu üretme problemi için azalt-bir-tekniği ne önerir? Bir küçük problem, tüm (n-1)! permütasyonunu üretmektir. Daha küçük problemin çözüldüğünü varsayarsak, n'yi n-1 elemanlı her permütasyonun elemanları arasında mümkün olan n pozisyonun her birine ekleyerek daha büyük olana bir çözüm elde edebiliriz.

Bu şekilde elde edilen tüm permütasyonlar farklı olacaktır (neden?) ve toplam sayıları n(n-1)! = n! olacaktır. Bu nedenle, 1,..., n'nin tüm permütasyonlarını elde edeceğiz. Üretilen permütasyonları ya soldan sağa ya da sağdan sola ekleyebiliriz. n'yi 12 ... (n-1)'e sağdan sola doğru hareket ettirerek eklemekle başlamak ve ardından 1,..., n-1'in yeni bir permütasyonunun işlenmesi gerektiğinde yön değiştirmek yararlı olur.

Eng: Abdulrahman Hamdi

Decrease-and-Conquer Algoritmaları

• **Permütasyon Üretme**

Bu yaklaşımın n=3 için aşağıdan yukarıya doğru uygulanmasına bir örnek Şekil'de verilmiştir.Bu permütasyon üretme yönteminin yararlarından biri, minimal değişiklik gereksinimini karşılamasıdır: her permütasyon, içindeki sadece iki elemanı değiştirerek hemen önceki permütasyondan elde edilebilir.(Tartışılan yöntem için, bu iki eleman her zaman birbirine bitişiktir.), (Şekil'de üretilen permütasyonlar için bunu kontrol edin.) Minimal değişiklik gereksinimi, hem algoritmanın hızı hem de permütasyonları kullanan uygulamalar için faydalıdır.

Örneğin, 3.4. gezgin satıcı problemini kapsamlı arama ile çözmek için şehirlerin permütasyonlarına ihtiyacımız vardı. Bu tür permütasyonlar minimal değişiklik algoritmasıyla üretilirse, yeni bir turun uzunluğunu, önceki turun uzunluğundan doğrusal zaman yerine sabit zamanda hesaplayabiliriz (nasıl?).

start	1		
insert 2 into 1 right to left	12	21	
insert 3 into 12 right to left	123	132	312
insert 3 into 21 left to right	321	231	213

FIGURE Generating permutations bottom up.

Eng: Abdulrahman Hamdi

Decrease-and-Conquer Algoritmaları

• **Permütasyon Üretme**
n elemanlı permütasyonların aynı sıralamasını, n'nin daha küçük değerleri için permütasyonları açıkça üretmeden elde etmek mümkündür. Bu, bir permütasyondaki her k elemanıyla bir yön ilişkilendirilerek yapılabilir. Bu tür bir yönü, söz konusu elemanın üzerine yazılmış küçük bir okla gösteriyoruz, örneğin, $\vec{3}\ \overleftarrow{2}\ \vec{4}\ \overleftarrow{1}$.

Bir okla işaretlenmiş permütasyonda k elemanının mobil olduğu, okunun kendisine bitişik daha küçük bir sayıyı göstermesi durumunda söylenir. Örneğin, $\vec{3}\ \overleftarrow{2}\ \vec{4}\ \overleftarrow{1}$ permütasyonu için, 3 ve 4 mobil iken 2 ve 1 mobil değildir. Mobil bir eleman kavramını kullanarak, permütasyonları üretmek için Johnson-Trotter algoritmasının aşağıdaki açıklamasını verebiliriz.

Eng: Abdulrahman Hamdi

Decrease-and-Conquer Algoritmaları

- **Permütasyon Üretme**

```
ALGORİTMA JohnsonTrotter(n)
// Giriş: Pozitif bir tamsayı n
// Çıkış: {1, ..., n} permütasyonlarının listesi
// İlk permütasyonu 1 2 ... n olarak başlat
permütasyonlar = [1, 2, ..., n]
oklar = [←, ←, ..., ←]  // Her elemanın yönünü
    saklayan oklar
While: P içinde hareketli bir eleman olduğu sürece
    En büyük hareketli eleman k'yı bul
    k'yı yöneldiği taraftaki komşu elemanla yer
    değiştir
    k'dan büyük olan tüm elemanların yönünü tersine
    çevir
    Yeni permütasyonu listeye ekle
    Döngü sonu
    Permütasyonlar listesini döndür
```

İşte n=3 için bu algoritmanın bir uygulaması, en büyük mobil eleman kalın olarak gösterilmiştir:

$\overleftarrow{1}\,\overleftarrow{2}\,\overleftarrow{3}$ $\overleftarrow{\mathbf{1}}\,\overleftarrow{3}\,\overleftarrow{\mathbf{2}}$ $\overleftarrow{3}\,\overleftarrow{1}\,\overleftarrow{2}$ $\overrightarrow{\mathbf{3}}\,\overleftarrow{2}\,\overleftarrow{\mathbf{1}}$ $\overleftarrow{2}\,\overrightarrow{3}\,\overleftarrow{1}$ $\overleftarrow{\mathbf{2}}\,\overleftarrow{1}\,\overrightarrow{\mathbf{3}}$

Bu algoritma, permütasyon üretmek için en verimli algoritmalardan biridir; permütasyon sayısıyla orantılı sürede, yani $\Theta(n!)$ içinde çalışacak şekilde uygulanabilir. Elbette, n'nin çok küçük değerleri dışında hepsi için korkunç derecede yavaştır; ancak, bu algoritmanın "hatası" değil, daha ziyade problemin hatasıdır: sadece çok fazla öğe üretmeyi ister. Johnson–Trotter algoritması tarafından üretilen permütasyon sıralamasının tam olarak doğal olmadığı iddia edilebilir; örneğin, n(n–1) ... 1 permütasyonu için doğal yer listedeki sonuncusu gibi görünüyor. Bu, permütasyonlar artan sırada—lexicographic sıra olarak da adlandırılır—listelenmiş olsaydı geçerli olurdu.

Eng: Abdulrahman Hamdi

Decrease-and-Conquer Algoritmaları

• **Permütasyon Üretme**

der — sayılar bir alfabenin harfleri olarak yorumlanırsa, bir sözlükte listelenecekleri sıra budur. Örneğin, n = 3 için,

123 132 213 231 312 321

Peki, $a_1 a_2 \ldots a_{n-1} a_n$an'den sonraki permütasyonu sözlük sırasına göre nasıl üretebiliriz? $a_{n-1} < a_n$ ise, ki bu tüm permütasyonların tam olarak yarısı için geçerlidir, bu son iki elemanı basitçe yer değiştirebiliriz. Örneğin, 123'ü 132 takip eder. an-1 > an ise, permütasyonun en uzun azalan sonekini buluruz $a_{i+1} > a_{i+2} > \ldots > a_n$ (ancak $a_i < a_{i+1}$); a_i 'yi, sonekte a_i 'den büyük olan en küçük elemanla değiştirerek artırırız; ve artan sıraya koymak için yeni soneki tersine çevirirız.

Örneğin, 362541'i 364125 takip eder. Kökenleri 14. yüzyıl Hindistan'ına kadar uzanan bu basit algoritmanın sözde kodu burada.

ALGORITHM *LexicographicPermute*(n)

//Generates permutations in lexicographic order
//Input: A positive integer n
//Output: A list of all permutations of $\{1, \ldots, n\}$ in lexicographic order
initialize the first permutation with $12 \ldots n$
while last permutation has two consecutive elements in increasing order **do**
  let i be its largest index such that $a_i < a_{i+1}$ //$a_{i+1} > a_{i+2} > \cdots > a_n$
  find the largest index j such that $a_i < a_j$ //$j \geq i+1$ since $a_i < a_{i+1}$
  swap a_i with a_j //$a_{i+1} a_{i+2} \ldots a_n$ will remain in decreasing order
  reverse the order of the elements from a_{i+1} to a_n inclusive
  add the new permutation to the list

Eng: Abdulrahman Hamdi

Decrease-and-Conquer Algoritmaları

• Bir kümenin tüm alt kümelerinin üretmek

Bu bölümde, bir kümenin tüm altkümelerini (güç kümesini) üretmek için kullanılan algoritmalar anlatılmaktadır. A = {a_1, ..., a_n} kümesinin 2^n adet altkümesi vardır. "Azalt-bir" yaklaşımıyla, bir kümenin altkümeleri iki gruba ayrılabilir: a_n elemanını içermeyenler ve içerenler. a_n'i içermeyen altkümeler, A'nın ilk n-1 elemanının altkümeleridir. Diğer grup, a_n'i bu altkümelere ekleyerek elde edilir.Bu yaklaşım, {a_1, a_2, a_3} gibi küçük kümeler için uygulanabilir. Ayrıca, doğrudan çözüm için, n elemanlı bir kümenin 2^n altkümesi ile birebir eşleme yapan yöntemler de kullanılabilir.

n	subsets							
0	$\varnothing$							
1	$\varnothing$	$\{a_1\}$						
2	$\varnothing$	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$				
3	$\varnothing$	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$	$\{a_3\}$	$\{a_1, a_3\}$	$\{a_2, a_3\}$	$\{a_1, a_2, a_3\}$

Eng: Abdulrahman Hamdi

Decrease-and-Conquer Algoritmaları

• **Bir kümenin tüm alt kümelerinin üretmek**
n uzunluğundaki tüm 2" bit dizisi b_1, ..., b_n. Böyle bir yazışma kurmanın en kolay yolu, bir alt kümeye $b_i = 1$ ise a_i'nin alt kümeye ait olduğu ve $b_i = 0$ ise a_i'nin ait olmadığı bit dizisini atamaktır. (Bölüm 1.4'te bit vektörleri fikrinden bahsetmiştik.) Örneğin, 000 bit dizisi üç elemanlı bir kümenin boş alt kümesine karşılık gelir, 111 kümenin kendisine, yani $\{a_1, a_2, a_3\}$'e karşılık gelir ve 110, $\{a_1, a_2\}$'yi temsil eder. Bu yazışma ile n uzunluğundaki tüm bit dizilerini, gerektiğinde uygun sayıda önde gelen 0 ile doldurularak 0'dan 2^n-1'e kadar ardışık ikili sayılar üreterek oluşturabiliriz. Örneğin, n = 3 durumu için

bit dizileri 000 001 010 011 100 101 110 111
alt kümeleri Ø $\{a_3\}$ $\{a_2\}$ $\{a_2, a_3\}$ $\{a_1\}$ $\{a_1, a_3\}$ $\{a_1, a_2\}$ $\{a_1, a_2, a_3\}$

Bit dizileri bu algoritma ile sözlükbilimsel sırada (0 ve 1'in iki sembollü alfabesinde) oluşturulsa da, alt kümelerin sırası doğal olmaktan çok uzak görünüyor. Örneğin, a_i içeren herhangi bir alt kümenin, a_1, ..., a_{j-1} içeren tüm alt kümelerden sonra listelendiği, üç elemanlı kümenin Şekil 4.10'daki listesinde olduğu gibi, sözde ezilmiş sıraya sahip olmak isteyebiliriz. Yukarıdaki bit dizisi tabanlı algoritmayı, ilgili alt kümelerin ezilmiş bir sıralamasını elde edecek şekilde ayarlamak kolaydır (bu bölümün alıştırmalarındaki Problem 6'ya bakın).

Decrease-and-Conquer Algoritmaları

• **Bir kümenin tüm alt kümelerinin üretmek**
Daha zorlu bir soru, bit dizilerini oluşturmak için, her birinin hemen öncekilerden yalnızca tek bir bit ile farklı olduğu minimal değişiklik algoritmasının olup olmadığıdır. (Alt kümelerin dilinde, her alt kümenin hemen öncekilerden tek bir elemanın eklenmesi veya silinmesi, ancak her ikisi de olmamasıyla farklı olmasını istiyoruz.). Bu sorunun cevabı evet. Örneğin, n = 3 için,

000 001 011 010 110 111 101 100

Elde edebiliriz. Bu tür bir bit dizisi dizisine ikili yansıtılmış Gray kodu denir. AT&T Bell Laboratuvarları'nda araştırmacı olan Frank Gray, dijital sinyallerin iletimindeki hataların etkisini en aza indirmek için 1940'larda yeniden icat etti (örneğin, [Ros07], s. 642-643'e bakın). Yetmiş yıl önce, Fransız mühendis Émile Baudot bu tür kodları kullandı.

Decrease-and-Conquer Algoritmaları

• **Bir kümenin tüm alt kümelerinin üretmek** telgrafçılıkta. İşte ikili yansıtılmış Gray kodunu özyinelemeli olarak oluşturan sözde kod.

ALGORITHM *BRGC*(n)

//Generates recursively the binary reflected Gray code of order n
//Input: A positive integer n
//Output: A list of all bit strings of length n composing the Gray code
if $n = 1$ make list L containing bit strings 0 and 1 in this order
else generate list $L1$ of bit strings of size $n - 1$ by calling *BRGC*($n - 1$)
    copy list $L1$ to list $L2$ in reversed order
    add 0 in front of each bit string in list $L1$
    add 1 in front of each bit string in list $L2$
    append $L2$ to $L1$ to get list L
return L

Algoritmanın doğruluğu, 2^n bit dizisi oluşturması ve hepsinin farklı olmasından kaynaklanmaktadır. Bu iddiaların her ikisinin de matematiksel tümevarımla kontrol edilmesi kolaydır. İkili yansıtılmış Gray kodunun döngüsel olduğuna dikkat edin: son bit dizisi, ilk bit dizisinden tek bir bit ile farklıdır.

Decrease-and-Conquer Algoritmaları

• **Bir kümenin tüm alt kümelerinin üretmek**

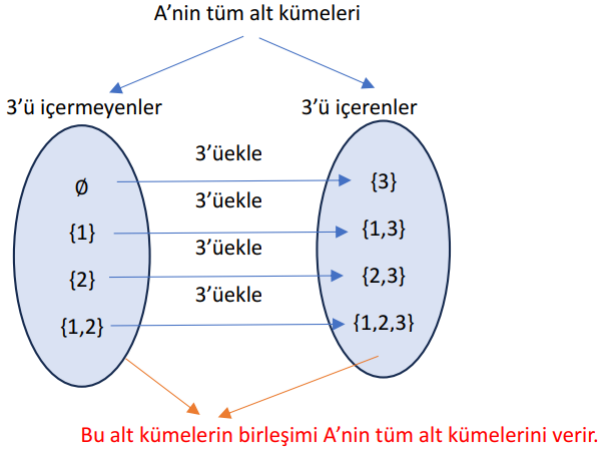
A kümesi n elemanlı bir küme olsun :

$$A = \{a_1, a_2, \ldots, a_n\}$$

A'nin an hariç diğer tüm elemanlarını içeren küme de B kümesi olsun. Bu durumda

$$B = \{a_1, a_2, \ldots, a_{n-1}\}.$$

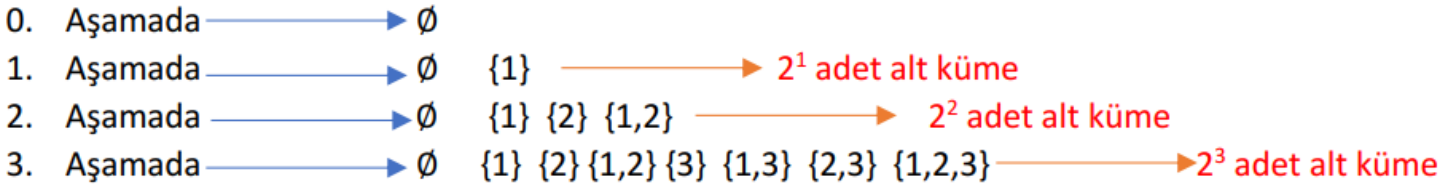
B'nin tüm alt kümeleri A'nin da bir alt kümesi olur. B'nin bu alt kümelerine an'i de eklersek , A'nin an'i içeren alt kümeleri elde edilir . an'i içeren bu alt kümeler ile an'i içermeyen alt kümeler birleştirildiğinde A'nin tüm alt kümeleri elde edilir. Bir ornek olarak $A\ \{1,2,3\}$ olsun:



Decrease-and-Conquer Algoritmaları

• Bir kümenin tüm alt kümelerinin üretmek

O halde $A = \{a_1, a_2, \ldots, a_n\}$ kümesinin tam alt kümelerini elde etmekse elimizdeki problem, biz bunu A'dan daha küçük bir küme olan $B = \{a_1, a_2, \ldots, a_{n-1}\}$kümesinin alt kümelerini bulma problemine düşürebiliriz/ indirgeyebiliriz . Eğer boyutu indirgenmiş bu problemi çözebilirsek (Yeni B'nin tüm alt kümelerini bulabilirsek) bu çözüme an 'i de ekleyip orijinal problem olan A'nin alt kümelerini bulma problemine erişebiliriz. Bu tıpkı an ' i hesaplama problemine benzer. Eğer an yerine bundan daha küçük an-1 problemini çözebilirsek, bu çözümü a ile çarpar bu mantıkla $A = \{1,2,3\}$ 'ün tüm alt kümelerini üretelim.



Decrease-and-Conquer Algoritmaları

- **Bir kümenin tüm alt kümelerinin üretmek**

  1 elemanlı bir küme için 2^1 adet alt küme üretiliyor
  2 elemanlı bir küme için 2^2 adet alt küme üretiliyor
  3 elemanlı bir tane için 2^2 adet alt küme üretiliyor.
  ⋮
  n elemanlı bir tane için 2^n adet alt küme üretiliyor.
  Şimdi bu şekilde altkümeleri elde etme problemini bir sözde kod olarak yazalım.

```
altKümelerUret(A[0, ..., n-1])
// Girdi: n elemanlı bir A dizisi
// Çıktı: A'nın elemanlarını içeren tüm
    altkümeler
alt_kümeler ⇐ {Ø}
    for i = 0'dan n-1'e
    for a in alt_kümeler // alt_kümeler'deki tüm
    mevcut altkümelere
        a ∪ A[i] kümesini alt_kümelere ekle
    return alt_kümeler
```

Şimdi bu algoritmanın zaman verimliliğini hesaplayalım.
i=0 iken içerdeki loop 1 kez çalışır (Ø'ye A[0]'ı ekler)
i=1 iken içerdeki loop 2 kez çalışır (Ø ve {A[0]}'a A[1]'i ekler)
i=2 iken içerdeki loop 4 kez çalışır (Ø, {A[0]}, {A[1]}, {A[0],A[1]}'e A[2]'i ekler)
⋮
i=n-1 iken içerdeki loop 2^{n-1} kez çalışır
toplam çalışma sayısı $1 + 2 + 2^2 + \ldots + 2^{n-1}$ bu bir geometrik seri toplamıdır.

$$1 + r + r^2 + \ldots + r^{n-1} \approx \frac{r^{n-1}}{r-1}$$

Burada r=2'dir. O halde $t(n) = 1 + 2 + \ldots + 2^{n-1} = 2^n - 1 \in \Theta(2^n)$

Eng: Abdulrahman Hamdi

Decrease-and-Conquer Algoritmaları

- **Bir kümenin tüm alt kümelerinin üretmek**

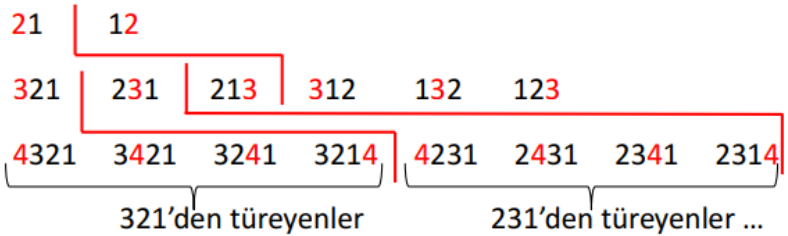
1'den n'ye kadar olan sayıların permütasyonları

n = 1 için permitasyonlar 1

n = 2 için permitasyonlar 21 12

n = 3 için permitasyonlar 1 321 231 213 312 132 123

n = 4 için permitasyonlar 1 4321 3421 3241 3214 4231 2431 2341 2314



Genel Kural: 1'den n'ye kadar permütasyonlar üretilirken, 1'den n-1'e kadar olan tüm permütasyonları bul. Bu permütasyonların her birine n defa n'yi ekle. Bu eklemeyi 1. , 2. , ... , n. Basamaklar için yap.

Decrease-and-Conquer Algoritmaları

Sahte Para Problemi(Fake Coin)
Sahte para tespiti probleminin çeşitli versiyonları arasında, burada sabit oranla azalma (decrease-by-a-constant-factor) stratejisini en iyi şekilde göstereni ele alıyoruz. Birbirine tamamen benzeyen n adet para arasında biri sahtedir. Bir terazi ile herhangi iki para grubunu karşılaştırabiliriz. Yani, terazi sağa ya da sola eğilerek ya da dengede kalarak, grupların ağırlıklarının aynı olup olmadığını ya da hangi grubun diğerinden daha ağır olduğunu (ne kadar farkla olduğunu değil) bize gösterebilir. Problem, sahte parayı tespit edecek verimli bir algoritma tasarlamaktır.

Bu problemin daha kolay bir versiyonu—ki burada onu tartışacağız—sahte paranın örneğin gerçek olanlardan daha hafif olduğu bilgisinin verildiği durumdur.[1]Bu problemi çözmek için en doğal fikir, n adet parayı ⌊n/2⌋ kadar iki gruba ayırmak (n tek ise bir parayı dışarıda bırakmak) ve bu iki grubu teraziye koymaktır.
Paraları terazide tartarız. Eğer iki yığın aynı ağırlıktaysa, kenara koyulan para sahte olmalıdır;aksi takdirde, sahte paranın daha hafif olduğu bilindiğinden, daha hafif olan yığınla aynı şekilde devam edebiliriz.

Eng: Abdulrahman Hamdi

Decrease-and-Conquer Algoritmaları

Sahte Para Problemi(Fake Coin)
Bu algoritmanın en kötü durumda ihtiyaç duyduğu tartım sayısı W(n) için kolayca bir özyinelemeli bağıntı kurabiliriz:
W(n) = W(⌊n/2⌋) + 1 (n > 1 için), başlangıç koşulu ise W(1) = 0'dır.Bu özyineleme size tanıdık gelmeli. Gerçekten de, bu ifade ikili aramada (binary search) en kötü durumda yapılan karşılaştırma sayısı için olanla neredeyse aynıdır. (Fark yalnızca başlangıç koşulundadır.)Bu benzerlik şaşırtıcı değildir, çünkü her iki algoritma da örnek boyutunu yarıya indirme tekniğine dayanır.

Bu özyinelemenin çözümü de ikili aramadakine çok benzer: $W(n) \approx \log_2 n$Bu noktada her şey oldukça açık ve belki sıkıcı görünebilir. Ama durun:Asıl ilginç olan, yukarıda açıklanan algoritmanın en verimli çözüm olmamasıdır.Daha verimli bir yöntem, paraları iki değil, yaklaşık n/3'lük üç yığına ayırmaktır.(Bunun kesin biçimi bu bölümün alıştırmalarında ele alınmıştır. Kaçırmayın! Eğer hocanız atlamışsa, Problem 10'u ödev olarak vermesini talep edin.)İki yığını tarttıktan sonra, örnek boyutunu üçte birine indirebiliriz. Bu doğrultuda, gereken tartım sayısının yaklaşık $\log_3 n$ olmasını beklemeliyiz ki bu da $\log_2 n$'den küçüktür.

Eng: Abdulrahman Hamdi

Decrease-and-Conquer Algoritmaları

Sahte Para Problemi(Fake Coin)

```
SahtePara(coinList):
    if coinList uzunluğu == 1 ise:
        return coinList[0] // Sahte para bulundu
        coinCount ← uzunluk(coinList)
    solGrup ← ilk ⌊coinCount / 2⌋ kadar para
    sagGrup ← kalan ⌊coinCount / 2⌋ kadar para
    kalanPara ← if coinCount tekse ortada kalan para
        sonuc ← Tart(solGrup, sagGrup)
        if sonuc == 'eşit':
        // Sahte para kalan paradır
        return kalanPara
    if sonuc == 'sol hafif':
        return SahtePara(solGrup)
    else:
        return SahtePara(sagGrup)
```

```
Tart(grup1, grup2):
    // Her iki grubun toplam ağırlığını hesapla
    // Gerçek uygulamada bu bir fiziksel terazi
    olurdu
    agirlik1 ← Toplam(grup1)
    agirlik2 ← Toplam(grup2)
        if agirlik1 == agirlik2:
        return 'eşit'
    if agirlik1 < agirlik2:
        return 'sol hafif'
    else:
        return 'sağ hafif'
```

Zaman Karmaşıklığı:

Her adımda n/2 boyutlu alt listeye geçildiğinden:

Rekürans: W(n) = W(⌊n/2⌋) + 1

Çözüm: W(n) = O(log n)

Eng: Abdulrahman Hamdi

Decrease-and-Conquer Algoritmaları

Rus Çiftçisi Çarpma Yöntemi(Russian Peasant Multiplication)

Şimdi iki pozitif tam sayının çarpımını hesaplamak için alışılmışın dışında bir algoritmayı inceleyeceğiz:Rus çarpma yöntemi ya da "multiplication à la russe" olarak bilinen bu yöntem.n ve m, çarpımını hesaplamak istediğimiz iki pozitif tam sayı olsun.Örnek büyüklüğünü n'in değeriyle ölçelim.Eğer n çift bir sayıysa, bu durumda n/2 ile ilgilenen daha küçük boyutta bir örnekle karşı karşıya kalırızve büyük örnekteki çözümü küçük örnekle ilişkilendiren açık bir formülümüz olur: $n * m = \frac{n}{2} * 2m$

Eğer n tek bir sayıysa, bu formülde yalnızca küçük bir düzeltme yapmamız gerekir:

$$n * m = \frac{n-1}{2} * 2m + m$$

Bu formülleri ve durma koşulu olarak 1 × m = m gibi basit durumu kullanarak, n × m çarpımını özyinelemeli (recursive) ya da yinelemeli (iterative) bir şekilde hesaplayabiliriz.Bu algoritmayla 50 × 65 çarpımının nasıl hesaplandığını gösteren bir örnek Şekil'de verilmiştir.Şekil a'daki parantez içinde gösterilen ek toplamların hepsinin, ilk sütundaki n değeri tek olan satırlarda yer aldığını fark edin.Bu nedenle, çarpımı, n sütununda tek olan satırların m sütunundaki değerlerini toplayarak bulabiliriz (bkz. Şekil b).

Eng: Abdulrahman Hamdi

Decrease-and-Conquer Algoritmaları

Rus Çiftçisi Çarpma Yöntemi(Russian Peasant Multiplication)

Ayrıca, bu algoritmanın yalnızca ikiye bölme (halving), ikiyle çarpma (doubling) ve toplama (adding) gibi basit işlemler içerdiğine dikkat edin—bu da örneğin, sınırlı işlem kabiliyetine sahip cihazlar için oldukça cazip bir özellik olabilir.

n	m	
50	65	
25	130	
12	260	(+130)
6	520	
3	1040	
1	2080	(+1040)
	2080	+(130 + 1040) = 3250

(a)

n	m	
50	65	
25	130	130
12	260	
6	520	
3	1040	1040
1	2080	2080
		3250

(b)

Eng: Abdulrahman Hamdi

çarpım tablosunu ezberlemek istemeyenler için cazip bir özellik olabilir.İşte bu özelliği, algoritmayı muhtemelen Rus çiftçileri için çekici kılan etken olmuştur.Batılı ziyaretçilere göre, bu yöntem 19. yüzyılda Rus çiftçileri tarafından yaygın olarak kullanılmıştır ve adını da buradan almıştır.(Aslında, bu yöntem M.Ö. 1650 gibi erken bir tarihte Mısırlı matematikçiler tarafından da biliniyordu. Ayrıca bu yöntem, donanımda çok hızlı bir şekilde uygulanabilir, çünkü ikili (binary) sayıların ikiye bölünmesi ve ikiyle çarpılması işlemleri bit kaydırma (shift) işlemleriyle yapılabilir.Ve bu işlemler, makine düzeyinde en temel işlemler arasında yer alır.

Decrease-and-Conquer Algoritmaları

Rus Çiftçisi Çarpma Yöntemi(Russian Peasant Multiplication)

```
RusKoyulusuCarpim (m, n)
//Girdi: m ve n tam sayıları
//Çıktı: m*n
toplam ← 0
While m ≠ 1 :
  If m mod 2 = 0
    m ← m / 2
  else
    m ← (m - 1) / 2
    toplam ← toplam + n
      n ← 2 * n
return toplam + n
```

Eng: Abdulrahman Hamdi

çarpım tablosunu ezberlemek istemeyenler için cazip bir özellik olabilir.İşte bu özelliği, algoritmayı muhtemelen Rus çiftçileri için çekici kılan etken olmuştur.Batılı ziyaretçilere göre, bu yöntem 19. yüzyılda Rus çiftçileri tarafından yaygın olarak kullanılmıştır ve adını da buradan almıştır.(Aslında, bu yöntem M.Ö. 1650 gibi erken bir tarihte Mısırlı matematikçiler tarafından da biliniyordu. Ayrıca bu yöntem, donanımda çok hızlı bir şekilde uygulanabilir, çünkü ikili (binary) sayıların ikiye bölünmesi ve ikiyle çarpılması işlemleri bit kaydırma (shift) işlemleriyle yapılabilir.Ve bu işlemler, makine düzeyinde en temel işlemler arasında yer alır.

Decrease-and-Conquer Algoritmaları

Josephus Problemi

Son örneğimiz, adını ünlü Yahudi tarihçi Flavius Josephus'tan alan Josephus problemi. Josephus, 66–70 yılları arasında Romalılara karşı Yahudi isyanına katılmış ve bu olayları yazıya dökmüştür. General olarak görev yaptığı dönemde, Jotapata kalesini 47 gün boyunca savunmuş; ancak şehir düşünce, kendisiyle birlikte 40 kişilik bir grupla bir mağaraya sığınmıştır.Oradaki isyancılar teslim olmaktansa ölmeyi tercih etmiştir. Josephus, sırayla herkesin yanındakini öldürmesini önerir ve bu sıranın kura ile belirlenmesini sağlar. Ancak kurnazca, sonuncu sırayı kendisine denk getirir. Mağarada hayatta kalan iki kişiden biri olarak, diğerini Romalılara teslim olmaya ikna eder.

Problem Tanımı:

1'den n'e kadar numaralandırılmış n kişi bir çemberde durmaktadır.Sayım, 1 numaralı kişiden başlar ve her ikinci kişi (yani 2, 4, 6, ...) elenir. Bu işlem, çemberde tek kişi kalana kadar devam eder.Amaç, başlangıçta hayatta kalan kişinin numarasını J(n) olarak belirlemektir.

Örnekler:

n = 6 için: İlk turda 2, 4, 6 elenir → sonra 3 ve 1 elenir → geriye 5 kalır, yani J(6) = 5

n = 7 için: İlk turda 2, 4, 6, 1 elenir → sonra 5 ve 3 elenir → geriye 7 kalır, yani J(7) = 7

Eng: Abdulrahman Hamdi

Decrease-and-Conquer Algoritmaları

Josephus Problemi

Analiz:

n sayısı çift ise, yani n = 2k:

İlk turda tüm çift numaralı kişiler elenir. Geriye k kişi kalır ve bu aslında aynı problemin daha küçük bir hali olur.Ancak numaralandırmalar değişmiştir. Yeni turda, eski pozisyonu bulmak için:

$J(2k) = 2 \times J(k) - 1$

n sayısı tek ise, yani n = 2k + 1:

Yine çift numaralı kişiler elenir.Hemen ardından 1 numaralı kişi de elenirse, yine k kişi kalır.Bu sefer pozisyonlar farklı şekilde geri dönüştürülür:

$J(2k + 1) = 2 \times J(k) + 1$

Başlangıç durumu:

$J(1) = 1$

Bu iki durumlu özyinelemeli formülden yola çıkarak, kapalı form (closed-form) çözüm de bulunabilir.Bunun için ileriye doğru birkaç değer hesaplanır, desen (pattern) keşfedilir ve matematiksel ispat yapılabilir.İlginç olan, bu çözümün en zarif hali n'nin ikilik (binary) gösterimi ile elde edilebilir:J(n), n'nin binary temsilinde 1-bit sola döngüsel kaydırılmasıdır.

Örnekler:

$J(6) = J(110_2) = 101_2 = 5 J(7) = J(111_2) = 111_2 = 7$

Yani, Josephus problemi sadece tarihi değil, aynı zamanda ikili sayı sistemleriyle de estetik bir şekilde bağlantılıdır.

Eng: Abdulrahman Hamdi

Algorithm Analysis And Design

Thanks for listening!

Eng: Abdulrahman Hamdi