Introduction to Algorithm Design

Decrease and Conquer

ROAD MAP

- **Decrease And Conquer**
  - Insertion Sort
  - Depth-First Search
  - Breadth-First Search
  - Topological Sorting
  - Algorithms For Generating Combinatorial Objects
  - Decrease By a Constant-Factor Algorithms
  - Variable-Size-Decrease Algorithms

Decrease And Conquer

Decrease and conquer tekniği

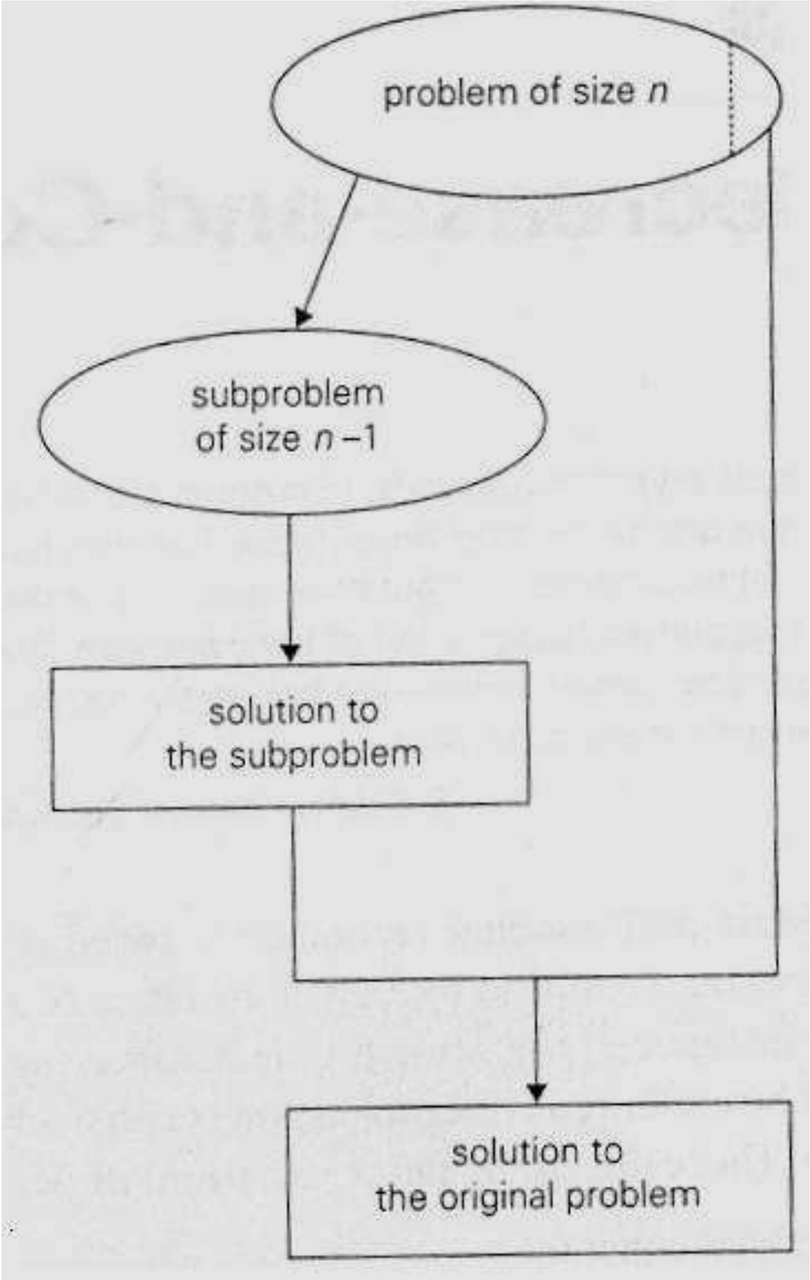
- Aynı problemin daha ufak boyutuna uygulanan çözümü kullanmaya dayanır.
- top down (recursively) ya da bottom up (recursive kullanmadan, **incremental yaklaşım**) ile çözülür
- Decrease ve conquer tekniğinin çeşitleri :

  1. Decrease by a constant (Sabit ile azaltma)
  2. Decrease by a constant factor (Sabit bir çarpan ile azaltma)
  3. Variable size decrease (Değişken Boyutta azaltma)

Decrease And Conquer

1. Decrease by a constant

- Her seferinde problemin boyutu sabit bir miktarda azalmaktadır.
  - Genelde bu sabit bire eşittir.



- **Decrease (by one) and conquer technique**

Decrease And Conquer

Örnek : pozitif bir tamsayı a için a^n'i hesaplamak.

- n boyutlu problem ve n-1 boyutlu problemin ilişkisi aşağıdaki formul ile elde edilmektedir.

$$a^n = a^{n-1}.a \qquad\qquad f(n) = a^n$$

- Top-down olarak aşağıdaki recursion ile hesaplanabilir.

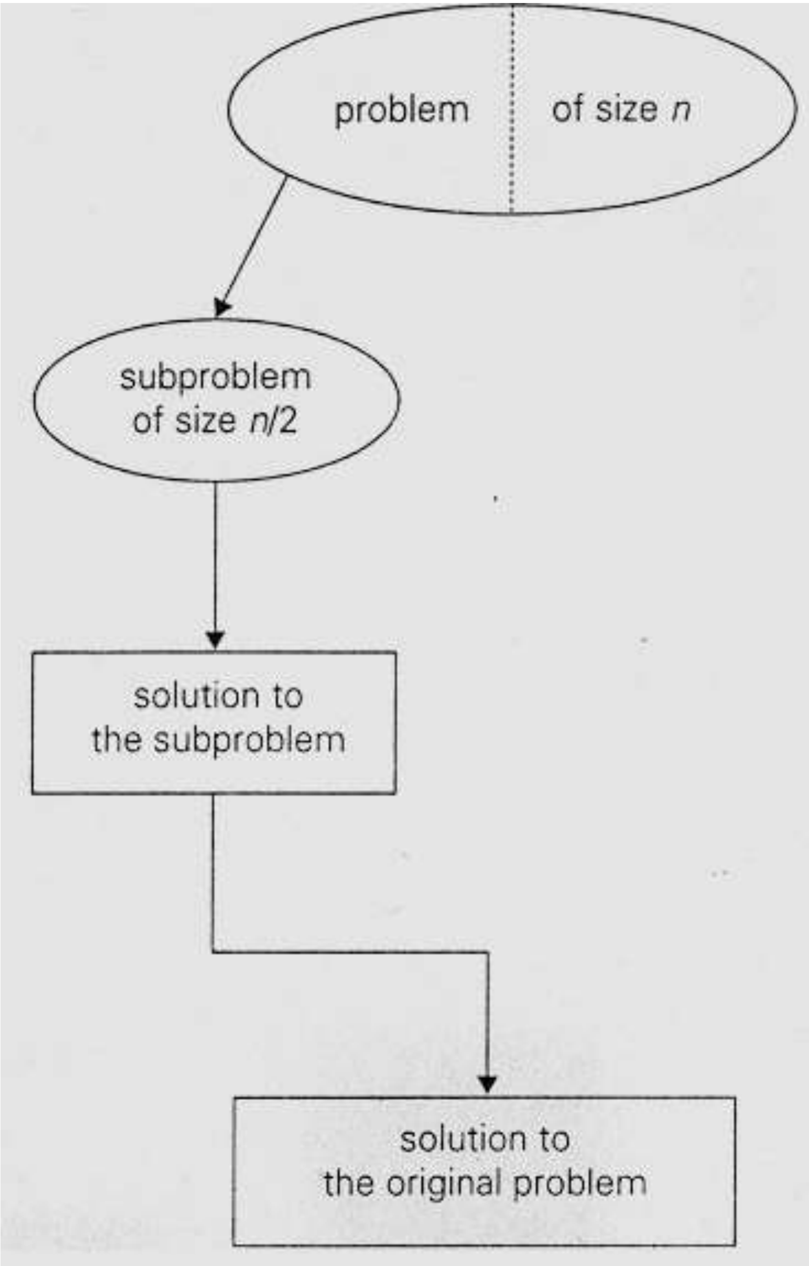
$$f(n) = \begin{cases} f(n-1).a & if \ \mathrm{n} > 1 \\ a & if \ \ n = 1 \end{cases}$$

- bottom up yaklaşımla *n-1* defa çarpma işlemi ile yapılabilir.
  - Bruteforce ile aynı

Decrease And Conquer

2. Decrease by a constant factor (Sabir bir katsayı ile azaltma)

- Problemlerin boyutunu her seferinde sabit bir oran ile azaltmak
  - Çoğu uygulamada bu 2’dir.



- **Decrease (by half) and conquer technique**

Decrease And Conquer

Örnek: pozifir bir tamsayı a için a^n'i hesaplamak

- a^n i hesaplayacaksak, $a^{n/2}$ 'i hesaplayabiliriz.

$$a^n = \left(a^{n/2}\right)^2$$

Bu yaklaşım tüm tamsayılar için çalışır mı?

Decrease And Conquer

Formül tek ve çift sayılar için farklıdır.

$$a^n = \begin{cases} \left(a^{n/2}\right)^2 & \text{if n is even and positive} \\ \left(a^{(n-1)/2}\right)^2 . a & \text{if n is odd and greater than 1} \\ a & \text{if n} = 1 \end{cases}$$

Bu algoritma *O(logn)'dir*

Decrease And Conquer

Bu algoritma divide & conquer algoritma yaklaşımından farklıdır.

$$a^n = \begin{cases} a^{\lfloor n/2 \rfloor} . a^{\lceil n/2 \rceil} & \text{if } n > 1 \\ a & \text{if } n = 1 \end{cases}$$

Bu yol etkin midir?

Decrease And Conquer

3. Variable size decrease (Değişken sayıda azalma)

- Problem boyutundaki azalma her iterasyonda farklı olmaktadır.
- Örn: İki tamsayının en büyük ortak bölenini bulan Euclid’s algoritma

  gcd (m,n) = gcd (n, m mod n)

  Sağ taraftaki parametre her zaman soldan daha küçük olmaktadır.

3 Types of Decrease and Conquer

- *Decrease by a constant* (genelde 1 azalma):
  - insertion sort
  - topological sorting
  - algorithms for generating permutations, subsets

- *Decrease by a constant factor* (genelde yarıya inme)
  - binary search vebisection method
  - exponentiation by squaring (Karesini alarak üs alma)
  - multiplication à la russe (Rus Çarpma Metodu)

- *Variable-size decrease*
  - Euclid’s algoritması
  - selection by partition
  - Nim-like games

ROAD MAP

- **Decrease And Conquer**
  - **Insertion Sort**
  - **Depth-First Search**
  - **Breadth-First Search**
  - **Topological Sorting**
  - Algorithms For Generating Combinatorial Objects
  - Decrease By a Constant-Factor Algorithms
  - Variable-Size-Decrease Algorithms

Sorting (Sıralama)

Definition :

Bir *A[0 .. n-1]* array'ini sıralama

Decrease-and-conquer tekniklerinden hangisi kullanılabilir?

1. Decrease by a constant
2. Decrease by a constant factor
3. Variable size decrease

Sorting

decrease-by-a-constant ile:

- **Yaklaşım:**
  - Problemin daha ufak boyutlu halini düşünelim: *A[0 .. n-2] arrayini sıralama*
    - $A[0] \leq A[1] \leq \ldots \leq A[n-2]$
  - Ufak problem çözüldüğünde
    - *n-1 boyutunda sıralı bir arrayimiz var*
  - Artık *A[n-1]* ‘i uygun yerine yerleştirmemiz lazım
  - Üç farklı şekilde yapabiliriz.

Insertion Sort

1. Sıralanmış arrayi soldan sağa *A[n-1]'den büyük ilk elemanı bulana kadar tara*

2. Sıralanmış arrayi sağdan sola *A[n-1]'den küçük veya eşit elemanı bulana kadar tara. (Bu algoritma insertion sort)*

3. Binary search ile *A[n-1] 'e uygun bir yer bul. (binary insertion sort)*

Insertion Sort

- Uygulaması
  - top down, recursively
  - bottom up iteratively
    - Daha verimli
  - Olarak gerçekleştirilebilir.
- Bottom up algoritma:
  - Loop: *A[1]* ile başlar ve *A[n-1]'deki elemana uygun yer bulma ile biter*
  - *A[i]* 'yi uygun yerine yerleştir.
    - Daha önceden sıralanmış ilk i elemanın içerisine yerleştir.

Insertion Sort

ALGORITHM *InsertionSort*($A[0..n-1]$)

//Sorts a given array by insertion sort
//Input: An array $A[0..n-1]$ of n orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 1$ **to** $n-1$ **do**
    $v \leftarrow A[i]$
    $j \leftarrow i-1$
    while $j \geq 0$ **and** $A[j] > v$ **do**
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j-1$
    $A[j+1] \leftarrow v$

Basic operation "A[j] > v" kıyaslamasıdır.

Why not j≥0 ?

Insertion Sort

- <u>Example :</u>

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

Insertion Sort

- **Analiz :**
  - Toplamda yapılan anahtar karşılaştırması inputun doğasına bağlıdır.
  - worst case

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

  - best case

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

  - average case

$$C_{avg}(n) \approx \frac{n^2}{4} \in \Theta(n^2)$$

Insertion Sort

- **Tartışma:**
  - En kötü durumda insertion sort, selection sort ile aynı sayıda kıyaslama yapar.
  - Sıralanmış arrayler için insertion sort'un performansı çok iyidir.
    - Nerdeyse sıralanmış arrayler üzerinde de iyi performansa sahiptir.(Bu şekilde uygulamalar vardır.).
  - ***shellsort*** isimli varyasyonu büyük dosyaları sıralamada iyi performans göstermektedşr

Topological Sorting

- **Tanım:**
  - Verilen bir yönlü çizgede (directed graph), bir vertex sırası istenmektedir.
  - where for every edge in the graph, the vertex where the edge starts is listed before the vertex where the edge ends
  - İçerisinde cycle olan çizgelerde bir çözüm yoktur.
    - Çizgenin *dag olması gerekmektedir (directed acyclic graph)*
  - Topological sortingin değişik alternatif çözümleri bulunmaktadır.

Topological Sorting

İki Algoritma:

- DFS Kullanımı
  - Vertexleri dead-end olma sırasının tersi ile
  - Eğer bir back edge varsa, çizge dag değildir
- CENG222'de görmüş olduğunuz algoritma
  - decrease-and-conquer tekniğine dayalı
  - Kendisine gelen(incoming) kenarı olmayan vertexi bulma

Topological Sorting

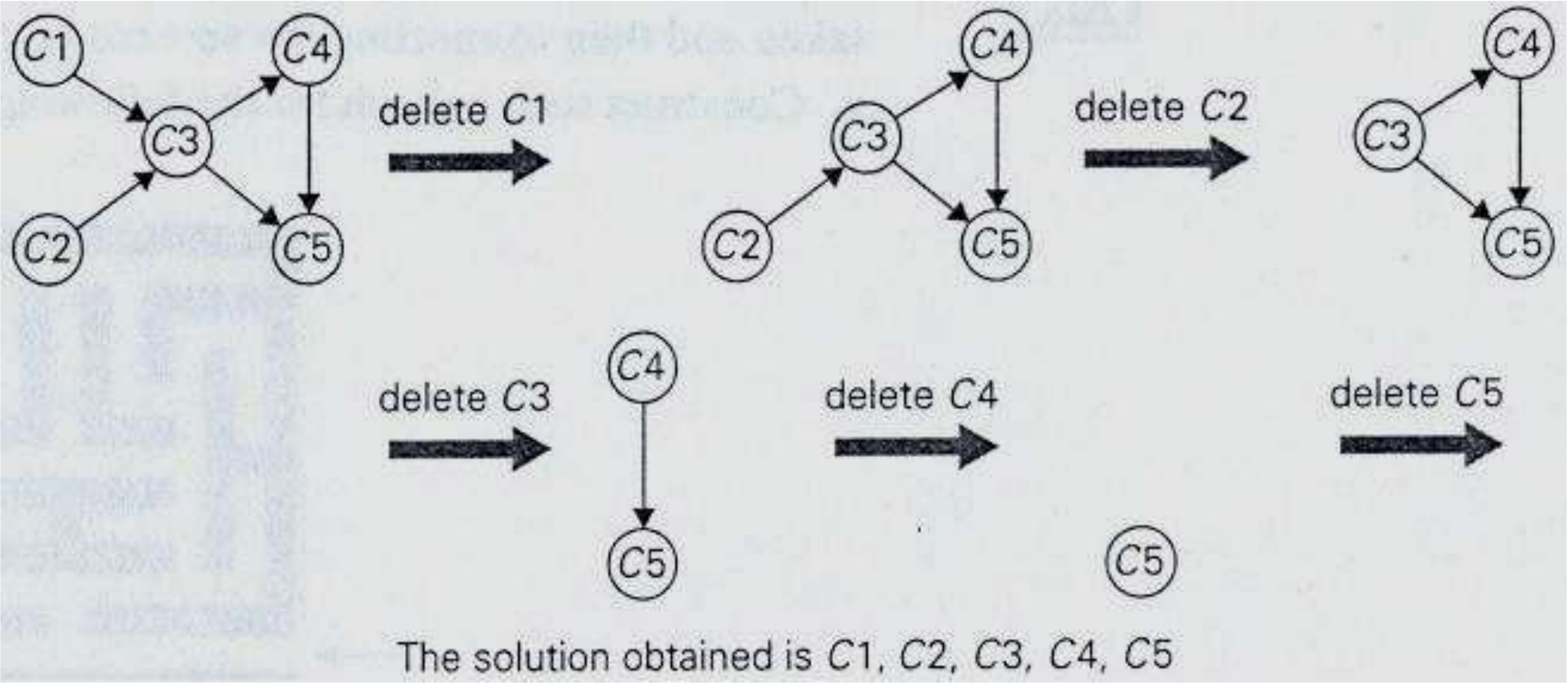


Illustration of the source removal algorithm for the topological sorting problem

ROAD MAP

- **Decrease And Conquer**
  - Insertion Sort
  - Depth-First Search
  - Breadth-First Search
  - Topological Sorting
  - **Algorithms For Generating Combinatorial Objects**
  - Decrease By a Constant-Factor Algorithms
  - Variable-Size-Decrease Algorithms

Kombinasyonla ilgili Nesneler Oluşturmak için Algoritmalar

- **Tanım:**
  - Kombinasyonla ilgili nesnelerin önemli türleri
    - permütasyonlar
    - Kombinasyonlar
    - Verilen bir kümenin alt kümeleri
  - Farklı seçeneklerin bulunduğu problemler içerisinde görülür
    - Önceden tartışılan (TSP, Knapsack)
  - Bu problemleri çözmek için kombinasyon nesnelerinin oluşturulması gerekmektedir.
    - Kaç tane oldukları ile ilgilenmiyoruz.

Permütasyon

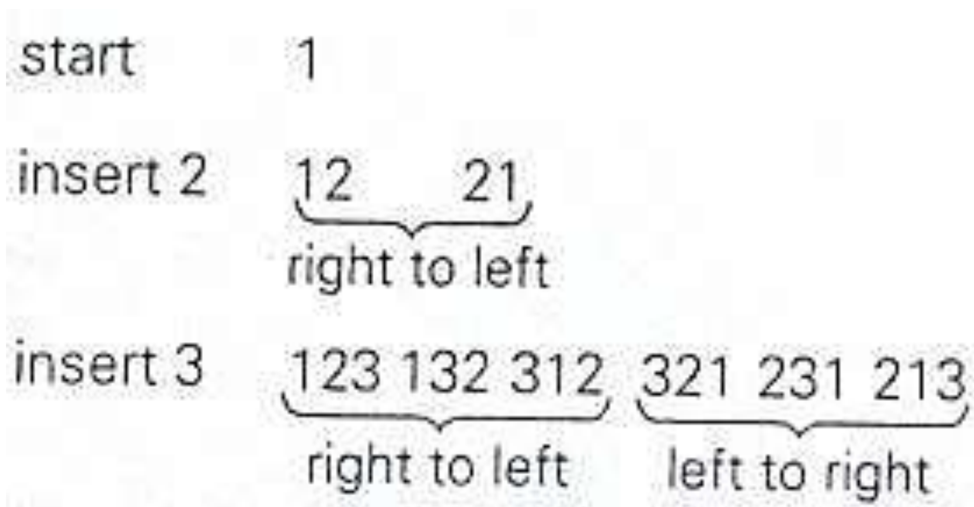
- 1'den n'e kadar olan tamsayıların permütasyonunu bulmamız gerekirse
  - N elemanlı bir kümenin $\{a_i, \ldots, a_n\}$
  - indisleri olarak düşünülebilir .
- Tüm n! Permütasyonların oluşması ile ilgili decrease-by-one tekniği ne olabilir ?

Permütasyonları Oluşturmak

- **Yaklaşım:**
  - Problemin 1 eleman azalmış hali (n-1) eleman için permütasyonları oluşturmaktır.
  - Bu daha küçük problemi çözdüğümüzü varsayalım
  - Problemin kendisini
  - n-1 elemanlı permütasyonlara ekleyerek
    - İki farklı şekilde ekleyebiliriz
  - Toplam permütasyon sayısı:
  - $n.(n-1)! = n!$

Permütasyonları Oluşturmak

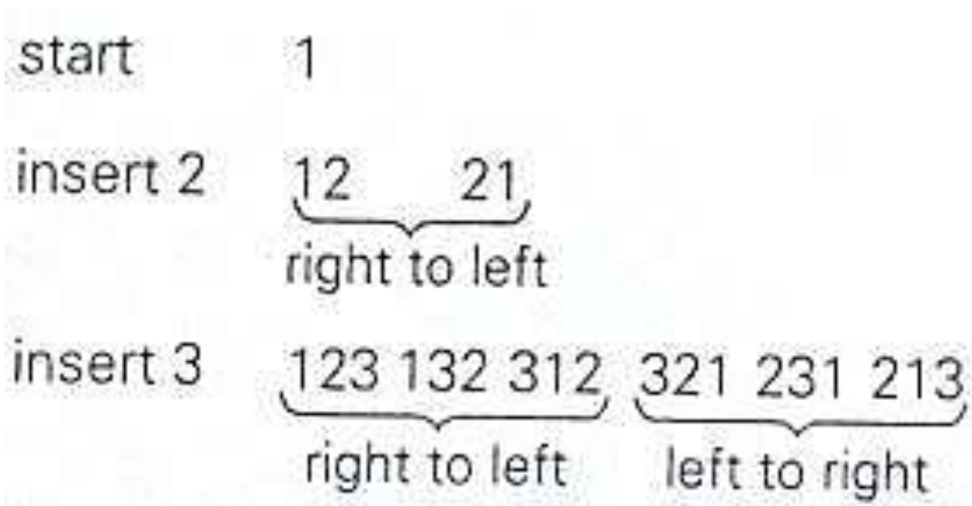
- n’i daha önceden oluşturulmuş permutasyonlara ekleyebiliriz.
  - Soldan sağa
  - Sağdan sola



Permütasyonları Oluşturmak

<u>Yaklaşım:</u>

- *{1,2,…,(n-1)} elemanlarının oluşturduğu tüm permütasyonların bulunduğunu varsayalım*
- n ‘yi *1,2,…,(n-1)* Permütasyonlarına sağdan sola ekle
- Her yeni n-1 elemanlı permütasyona eklemede ekleme yönünü değiştir.



Permütasyonları Oluşturmak

- Bu sıra ***minimal-change*** ihtiyacını karşılamaktadır
  - Her oluşturulan permütasyon bir önceki permütasyonun iki elemanın swap edilmesi ile elde edilebilir.
  - Algoritmanın hızı açısından yararlıdır.

  Örn: TSP'deki avantajı:
  - Yeni turun toplam uzunluğu iki adımda hesaplanabilir.
    - Bir önceki turun uzunluğu kullanılarak.
- Bu yaklaşım {*1,2,…,(n-1)*} *elemanlarının tüm permütasyonlarının hazır bir şekilde bulunmasını istemektedir.*
  - *Yüksek depolama alanı gerektitirir.*

Permütasyonları Oluşturmak

Aynı sırayı daha başka bir yöntemle elde edebilirsiniz:

- Permütasyondaki her bir elemana bir yön vermelisiniz
- Yön için küçük bir ok kullanabilirsiniz

$$\overrightarrow{3}\ \overleftarrow{2}\ \overrightarrow{4}\ \overleftarrow{1}$$

- k. eleman mobildir:
  - Eğer kendi oku kendisinden küçük bir elemanı işaret ediyorsa
  - 3 ve 4 mobildir.
  - 2 ve 1 mobil değildir.
- Biraz sonra bahsedeceğimiz algoritma bu notasyonu kullanmaktadır.

Permütasyonları Oluşturmak

```
ALGORITHM  Johnson Trotter (n)
 // Implements Johnson-Trotter algorithm for generating
   permutations
 // Input  : A positive integer n
 // Output : A list of permutations of {1, … , n}

 Initialize the first permitation with ←1 ←2 … ←n
 while there exists a mobile integer k do
   find the largest mobile integer k
    swap k and the adjacent integer its arrow points to
    reverse the direction of all integers that are larger than k
```

Permütasyonları Oluşturmak

- ***Johnson Trotter*** algoritmasının çalıştırılması:

← ← ←	← ← ←	← ← ←	→ ← ←	← → ←	← ← →
1 2 **3**	1 **3** 2	3 1 **2**	**3** 2 1	2 **3** 1	2 1 3

Permütasyonları Oluşturmak

- **Analiz:**

Algoritmanın çalışma maliyeti $\Theta(n!)$*'dir.*

Optimal midir?

Permütasyonları Oluşturmak

- **Tartışma:**
  - ***Johnson Trotter*** algoritması permütasyon oluşturmak için kullanılan en etkin algoritmalardan biridir.
  - Aslında, büyük n değerleri için algoritma aşırı derecede yavaş çalışır.
  - Bu durum algoritmanın değil problemin doğasından kaynaklanmaktadır.
    - Çok sayıda eleman oluşturulması istenmektedir.

Permütasyonları Oluşturmak

- 14. yüzyıl hindistan

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

Alt Küme Oluşturmak

- Knapsack Problemini Hatırlayalım
  - Knapsack problemi verilen sırt çantasının taşıyabileceği en değerli alt kümenin bulunmasını istemektedir.
- Exhaustive search tüm alt kümelerin oluşturulmasını istemekteydi.
- *$A = \{a_1, a_2, \ldots, a_n\}$ kümesinin tüm alt kümelerini oluşturmak isteyeceğiz.*

- Bu problemi decrease-by-one yaklaşımı ile nasıl çözmeliyiz?

Alt Küme Oluşturmak

- $A = \{a_1, a_2, \ldots, a_n\}$ kümesinin tüm alt kümeleri ikiye bölünebilir.
  - a_n'i içeren ve a_n'i içermeyen kümeler
- Önce $\{a_1, a_2, \ldots, a_{n-1}\}$ *tüm alt kümelerini oluştururuz ardından,*
- Önceki adımda elde ettiğimiz tüm alt kümelere *a_n'i ekleyebiliriz. Sonuçta $\{a_1, a_2, \ldots, a_n\}$ kümesinin tüm alt kümelerine erişebiliriz.*
- Yaklaşım pratik değildir.
  - *$\{a_1, a_2, \ldots, a_{n-1}\}$ kümesinin tüm alt kümelerini bulmamız gerekiyor.*
- Daha farklı yaklaşımlar da vardır.

Alt Küme Oluşturmak

- Generating subsets bottom up

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

Alt Küme Oluşturmak

- Fikir:
  - Bir b bit stringi oluştur. Bu bir string'inde:
    - $b_i = 1$, a_i kümenin elemanı ise
    - $b_i = 0$, a_i kümenin elemanı değilse
  - For set of a three-elements $\{a_1, a_2, a_3\}$

bit strings	000	001	010	011	100	101	110	111
subsets	$\emptyset$	$\{a_3\}$	$\{a_2\}$	$\{a_2, a_3\}$	$\{a_1\}$	$\{a_1, a_3\}$	$\{a_1, a_2\}$	$\{a_1, a_2, a_3\}$

  - Alt- kümeleri oluşturmak için tüm bit stringleri oluştur.
    - 0 'dan 2^n-1 e kadar tüm sayıları temsil edecek bit stringlerini oluştur.

Generating Subsets

- Önceki yöntemle bit stringleri *lexicographic sıraya göre oluşmaktadır.*

  000 001 010 011 100 101 110 111

- Aynı zamanda her her alt kümenin bir öncekinden bir bit farklı olduğu algoritma da mevcuttur(minimal change).

  000 001 011 010 110 111 101 100

- Örn: ***binary reflected Gray code***
- Gray code çoğu uygulamada yararlı olacak bir çok özelliği vardır.

Binary Reflected Gray Code

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

ROAD MAP

- **Decrease And Conquer**
  - Insertion Sort
  - Depth-First Search
  - Breadth-First Search
  - Topological Sorting
  - Algorithms For Generating Combinatorial Objects
  - **Decrease By a Constant-Factor Algorithms**
  - Variable-Size-Decrease Algorithms