

Rekabetçi Programcının El Kitabı

Antti Laaksonen

19 Ağustos 2019

Çeviri Tarihi: 21 Ağustos 2022

Çevirenler: Arda Kaz ve Alkın Kaz

İçindekiler

Çevirmenin Ön Sözü	ix
Ön Söz	xi
I Basit Teknikler	1
1 Başlangıç	3
1.1 Kodlama Dilleri	3
1.2 Girdi ve Çıktı	4
1.3 Sayılarla Çalışmak	6
1.4 Kodu Kısaltmak	8
1.5 Matematik	10
1.6 Yarışmalar ve Kaynaklar	15
2 Zaman Karmaşıklığı	19
2.1 Hesaplama Kuralları	19
2.2 Karmaşıklık Sınıfları (Complexity classes)	22
2.3 Verimliliği Tahmin Etme	23
2.4 En Büyük Alt dizi Toplamı	23
3 Sıralama	27
3.1 Sıralama Teorisi	27
3.2 C++ Dilinde Sıralama	32
3.3 İkili Arama Algoritması (Binary Search)	34
4 Veri Yapıları	39
4.1 Dinamik Diziler	39
4.2 Küme (Set) Yapıları	41
4.3 Map Yapıları	42
4.4 İteratörler ve Aralıklar (Iterators and Ranges)	43
4.5 Diğer Yapılar	46
4.6 Comparison to sorting	49
5 Tam Arama (Complete Search)	53
5.1 Alt küme Oluşturmak (Generating Subsets)	53
5.2 Permütasyon Oluşturmak (Generating Permutations)	55

5.3	Geri İzleme (Backtracking)	56
5.4	Aramayı Budama (Pruning The Search)	57
5.5	Ortada Buluşmak (Meet in the middle)	60
6	Açgözlü Algoritmalar (Greedy Algorithms)	63
6.1	Para Problemi (Coin problem)	63
6.2	Zaman Planlaması (Scheduling)	64
6.3	Görevler ve Son Teslimler (Tasks and Deadlines)	66
6.4	Toplamları Küçültmek (Minimizing Sums)	67
6.5	Veri Sıkıştırma (Data Compression)	68
7	Dinamik Programlama	71
7.1	Para Problemi	71
7.2	En Uzun Artan Altdizi (Longest Increasing Subsequence)	76
7.3	Düzlemde Yollar	77
7.4	Knapsack Problemleri	78
7.5	Değişiklik Farkı (Edit Distance)	80
7.6	Fayans Sayma (Counting Tilings)	81
8	Amortize Analizi	83
8.1	İki İşaretçi Metodu (Two Pointers Method)	83
8.2	En Yakın Küçük Elemanlar (Nearest Smaller Elements)	85
8.3	Sürgülü Pencere Minimumu (Sliding Window Minimum)	87
9	Aralık Sorguları	89
9.1	Statik Dizi Sorguları	90
9.2	İkili İndisli Ağaç (Binary Indexed Tree)	92
9.3	Segment Ağacı	95
9.4	Ek Teknikler	99
10	Bit Manipülasyonu	101
10.1	Bit Gösterimi	101
10.2	Bit Operasyonu	102
10.3	Kümeleri Göstermek	104
10.4	Bit Optimizasyonu	106
10.5	Dinamik Programlama	108
II	Çizge Algoritmaları	113
11	Çizgenin Temelleri	115
11.1	Çizge Terminolojisi (Graph Terminology)	115
11.2	Çizge Gösterimi (Graph Representation)	119
12	Çizgede Dolaşma (Graph Traversal)	123
12.1	Derinlik Öncelikli Arama (Depth-First Search)	123
12.2	Genişlik Öncelikli Arama (Breadth-First Search)	125
12.3	Uygulamalar	127

13 En Kısa Yolu Bulmak (Shortest Paths)	131
13.1 Bellman–Ford Algoritması	131
13.2 Dijkstra’nın Algoritması	134
13.3 Floyd–Warshall Algoritması	137
14 Ağaç Algoritmaları	141
14.1 Tree Dolaşımı	142
14.2 Çap	143
14.3 Bütün En Uzun Yollar (All Longest Paths)	145
14.4 İkili Ağaç (Binary Tree)	147
15 Kapsayan Ağaç (Spanning Trees)	149
15.1 Kruskal’ın Algoritması	150
15.2 Union-find Yapısı	153
15.3 Prim’in Algoritması	155
16 Yönlü Çizgeler	157
16.1 Topolojik Sıralama (Topological Sorting)	157
16.2 Dinamik Programlama	159
16.3 Ardıl Yollar (Successor Paths)	162
16.4 Döngü Bulma	163
17 Güçlü Bağlanırlık (Strong connectivity)	165
17.1 Kosaraju’nun Algoritması (Kosaraju’s Algorithm)	166
17.2 2SAT Problemi	168
18 Ağaç Sorguları	171
18.1 Ataları Bulmak	171
18.2 Alt Ağaçlar ve Yollar	172
18.3 En Yakın Ortak Ata (Lowest Common Ancestor (LCA))	175
18.4 Çevrimdışı Algoritmalar	178
19 Yollar ve Devreler	183
19.1 Euler Yolu	183
19.2 Hamilton Yolları	187
19.3 De Bruijn Dizileri	188
19.4 Atın Turları (Knight’s Tours)	189
20 Akışlar (Flows) ve Kesimler (Cuts)	191
20.1 Ford–Fulkerson Algoritması	192
20.2 Ayırık Yollar	196
20.3 Maksimum Eşleşme	197
20.4 Yol Kaplaması	200

III İleri Konular	205
21 Sayılar Teorisi	207
21.1 Asal sayılar(Primes) ve Bölenler(Factor)	207
21.2 Modüler Aritmetik	211
21.3 Eşitlikleri Çözmek	214
21.4 Diğer Sonuçlar	215
22 Kombinatorik (Combinatorics)	217
22.1 Binom Katsayıları (Binomial Coefficients)	218
22.2 Katalan Sayıları	220
22.3 İçerme-Dışarma (Inclusion-Exclusion)	222
22.4 Burnside’ın Lemma’sı	224
22.5 Cayley’in Formülü	225
23 Matris (Matrix)	229
23.1 Operasyonlar	229
23.2 Linear Tekrar (Linear Recurrences)	232
23.3 Çizgeler ve Matrisler	234
24 Olasılık (Probability)	237
24.1 Hesaplama	237
24.2 Olaylar	238
24.3 Rastgele Değişkenler	240
24.4 Markov Zincirleri	242
24.5 Rastgele Algoritmalar	243
25 Oyun Teorisi (Game Theory)	247
25.1 Oyun Durumları	247
25.2 Nim Oyunu	249
25.3 Sprague–Grundy Teoremi	250
26 Yazı Algoritmaları	255
26.1 Yazı Terminolojisi	255
26.2 Trie Yapısı	256
26.3 Yazı Hashleme	257
26.4 Z-Algoritması	259
27 Karekök Algoritmaları	263
27.1 Algoritmaları Birleştirme (Combining Algorithms)	264
27.2 Sayı Bölmesi (Integer Partitions)	266
27.3 Mo’nun Algoritması (Mo’s Algorithm)	267
28 Segment Ağacının Devamı	269
28.1 Lazy propagation	270
28.2 Dinamik Ağaçlar	273
28.3 Veri Yapıları	275
28.4 İki-Boyutluluk	276

29 Geometri	277
29.1 Karmaşık Sayılar	278
29.2 Noktalar ve Çizgiler	280
29.3 Çokgen Alanı	283
29.4 Uzaklık Fonksiyonları	284
30 Çizgi Süpürme Algoritmaları (Sweep Line Algorithms)	287
30.1 Kesişim Noktaları	288
30.2 En Yakın Çift Problemi	289
30.3 Convex hull Problemi	290
Kaynakça	293
Dizin	299

Çevirmenin Ön Sözü

Bu kitap, yazılım ve bilgisayar bilimleri alanında dilimizde yeni oluşmaya başlayan literatüre bir katkıda bulunma hedefiyle çevrilmiştir.

Bilim olimpiyatları camiasında sıklıkla yinelenen bir nokta olan Türkçe kaynak eksikliğine ivedi bir çözüm sunma amacıyla çevirdiğimiz bu kitap, umuyoruz ki hem lise (IOI) hem de üniversite (ICPC) seviyesinde ulusal ve uluslararası yarışmalara hazırlanmada gencimize büyük ölçüde yardımcı olacaktır.

Alandaki bazı terimlerin Türkçe genelgeçer bir tanım yoksunluğu durumunda orijinalleri terimlerin ilk bahsinde parantez içinde verilerek çevrilmiştir.

Çeviri hakkındaki yorum ve önerilerinizi arda.kaz@hotmail.com adresine gönderebilirsiniz.

Çevirmenler olarak tutarlılık konusunda ciddi bir hassasiyet göstermemize rağmen olası çeviri hataları ya da anlaşılmayan noktalarda [metnin orijinaline](#) danışılmalıdır. Orijinal metin ile çeviri metin arasındaki anlam farklılıklarından doğabilecek herhangi bir sorunda bu kitapçığın çevirisi, basımı ve dağıtımı başta olmak üzere herhangi bir şekilde oluşumunda yer almış kimseler sorumlu tutulamaz.

İstanbul, Ağustos 2022
Arda Kaz ve Alkın Kaz

Ön Söz

Bu kitabın amacı, rekabetçi (competitive) programlamaya kapsamlı bir giriş sunmaktır. Rekabetçi programlama hakkında herhangi bir arkaplan gereksinimi yoktur ancak okuyucunun temel programlama bilgilerine hakim olduğu varsayılmıştır.

Bu kitabın hedef kitlesi, algoritma öğrenmek isteyen ve belki de ileride Uluslararası İformatik Olimpiyatı (IOI, International Olympiad in Informatics) veya Uluslararası Üniversitelerarası Programlama Yarışması (ICPC, International Collegiate Programming Contest) gibi yarışmalara katılacak öğrencilerdir. Bununla beraber tabii ki de kitap, rekabetçi programlamaya ilgi duyan herkese uygundur.

Güçlü bir rekabetçi programcı olmak uzun bir zaman ister ancak bu yolculuk aynı zamanda bolca bilgi edinebileceğiniz bir öğrenme fırsattır. Eğer kitabı okumaya, problem çözmeye ve yarışmalara katılmaya yeterli zaman ayırırsanız algoritmalar hakkında genel itibariyle güçlü bir anlayış kazanacağınızdan şüpheniz olmasın.

Kitap, devamlı biçimde gelişmekte ve güncellenmektedir. Kitap hakkındaki geribildirimlerinizi ahslaaks@cs.helsinki.fi adresine gönderebilirsiniz.

Helsinki, Ağustos 2019
Antti Laaksonen

Kısım I

Basit Teknikler

Bölüm 1

Başlangıç

Rekabetçi programlama iki konudan oluşur: uygun algoritmayı bulmak (algoritmanın dizaynı) ve uygun algoritmanın koda geçirilmesi (implementasyonu).

Uygun algoritmayı bulmak (dizayn) için soru çözmek ve matematiksel düşünme gerekir.

Soruların analiz edilip yaratıcı bir şekilde çözülmesi önemlidir. Soruyu çözen algoritmanın hem doğru hem de verimli olması gerekir. Zaten genel olarak soruların temelinde verimli algoritmayı bulmak vardır.

Rekabetçi programcıların algoritmalar hakkında teorik bilgiye sahip olması gerekir. Tipik bir soru çözümü genelde bilinen tekniklerle yeni gözlemlerin birleşimidir. Rekabetçi programlamada çıkan teknikler aynı zamanda algoritmaların araştırma bazlı kısmının da temelini oluşturur...

Algoritmaların koda geçirilmesi (implementasyon) içinse iyi kodlama bilgisi gerekir. Rekabetçi programlamada çözümler belirli test caseler kullanılarak puanlanır. Bu yüzden sadece algoritmayı düşünerek bulmak yetmez, aynı zamanda bunun koda doğru bir şekilde geçirilmesi önemlidir.

Yarışmalarda yazılan kodların kısa ama aynı zamanda anlaşılabilir olması gerekir. Yarışmalarda verilen zamanın kısıtlı olması nedeniyle çözümlerin hızlı yazılması gerekir. Klasik yazılım mühendisliğinin aksine, çözümler kısa olup (çoğunlukla en fazla birkaç yüz satır kod) yarışma sonrası geliştirilmesi gerekmemektedir.

1.1 Kodlama Dilleri

Şu anda rekabetçi programlamada en çok kullanılan kodlama dilleri C++, Python ve Java'dır. Örneğin Google Code Jam 2017'de yarışmacıların ilk 3000'ünün 79 %'u C++, 16 %'sı Python ve 8 %'i Java kullanmıştır [29]. Bazı yarışmacılar birden çok kodlama dilini kullandılar.

Çoğu yarışmacı C++ dilini rekabetçi programlama için en iyi dil olarak görüyor ve C++ neredeyse her yarışma sisteminde bulunmaktadır¹. C++'ın yararları arasında çok hızlı ve verimli bir dil olmasıyla beraber çeşitli data yapıları ile algoritmaları kapsayan bir kütüphaneye sahip olması yer alır.

¹Çevirmen Notu (Ç.N.): TÜBİTAK Bilim Olimpiyatları'nda sadece C/C++ kullanılabilir.

Yine de birkaç dilde uzmanlaşıp onların yararlarını bilmekte fayda var. Örneğin soruda çok büyük sayılar gerekiyorsa Python, büyük sayılar için işlemleri halihazırda built-in bulundurmasından dolayı uygun bir seçenek olabilir. Neyse ki yarışmalardaki çoğu soru, herhangi bir kodlama dilinin avantajı olmayacak şekilde hazırlanmaktadır.

Bu kitaptaki örnek çözümler C++ ile yazılmış olup standart kütüphanedeki algoritma ve data yapıları sıklıkla kullanılmıştır. Çözümler C++11 formatında yazılmıştır ki bu format şu anki çoğu yarışmada kullanılabilir. Eğer C++ bilmiyorsanız, şimdi öğrenmek için iyi bir zaman olabilir.

C++ Kod Örneği

Klasik bir C++ kodu aşağıdaki gibi görünür.

```
#include <bits/stdc++.h>

using namespace std;

int main() {
    // cozum burada yazilir.
}
```

Kodun başındaki `#include` satırı, g++ derleyicisinin bir özelliği olup standart kütüphaneyi kodumuza eklememizi sağlar. Böylece `iostream`, `vector`, `algorithm`, gibi kütüphaneleri elle teker teker yüklemek yerine hepsini otomatik bir şekilde eklemiş oluruz.

`using` satırı, standart kütüphanedeki sınıfların ve fonksiyonların herhangi bir indikatör koymadan direkt olarak kullanılabilmesini söyler. Eğer `using` ifadesini kullanmazsak `cout` ifadesini `std::cout` şeklinde yazmamız gerekir.

Kod aşağıdaki komutla derlenebilir.

```
g++ -std=c++11 -O2 -Wall test.cpp -o test
```

Bu komut, `test.cpp`'den `test` adlı bir binary dosyası oluşturur.

Bu derleyici C++11'i kullanıp (`-std=c++11`) kodu optimize edip (`-O2`) olası hatalar hakkında uyarı verir. (`-Wall`).

1.2 Girdi ve Çıktı

Çoğu yarışmalarda girdi ve çıktı almak için klasik fonksiyonlar kullanılır. C++'da bu klasik fonksiyonlar girdi için `cin` ve çıktı için `cout`'dur. Bunla beraber C fonksiyonları olan `scanf` ve `printf` de kullanılabilir.

Program için olan girdiler genel olarak birbirlerinden boşluk veya yeni satır karakterleriyle ayrılmış sayılar ve stringlerdir. Bu girdiler `cin` ifadesiyle aşağıdaki gibi alınabilir:

```
int a, b;
```



```
string x;  
cin >> a >> b >> x;
```

Girdide her element arası yeni satır ve boşluk olduğu sürece bu tip bir kod her zaman çalışacaktır. Örneğin yukarıdaki kod aşağıdaki her iki girdiyi de düzgün şekilde okuyabilir.

```
123 456 monkey
```

```
123    456  
monkey
```

cout ifadesi çıktıları için aşağıdaki şekilde kullanılır.

```
int a = 123, b = 456;  
string x = "monkey";  
cout << a << " " << b << " " << x << "\n";
```

Bazen girdi ve çıktılar programı yavaşlatacak birer darboğaz (bottleneck) halini alabilir. Aşağıdaki iki satırı kodu programa eklemek girdi ve çıktıyı daha verimli hale getirir.

```
ios::sync_with_stdio(0);  
cin.tie(0);
```

Yeni satır “\n” ifadesinin endl’e göre daha hızlı çalıştığına dikkat edin çünkü endl her zaman flush operasyonu uygular..

C fonksiyonları scanf ve printf de alternatif olarak kullanılabilir. Bu fonksiyonlar genel olarak biraz daha hızlı fakat kullanımları daha zordur. Aşağıdaki kod, C fonksiyonlarını kullanarak iki tamsayı okur:

```
int a, b;  
scanf("%d %d", &a, &b);
```

Aşağıdaki kod iki tamsayı yazdırır:

```
int a = 123, b = 456;  
printf("%d %d\n", a, b);
```

Bazen programın içinde boşluk içermeye ihtimali olan bütün bir satırı okuması gerekir. Bu durumda aşağıdaki gibi getline fonksiyonu kullanılır:

```
string s;  
getline(cin, s);
```

Eğer girdinin miktarı bilinmiyorsa aşağıdaki gibi bir döngü yararlı olabilir:

```
while (cin >> x) {  
    // kod
```

```
}
```

Bu döngü, girdide okunmamış eleman kalmayana kadar elemanları okumaya devam eder.

Bazı yarışmalarda, girdi ve çıktılar için dosya kullanılır. Bu durumda kolay bir çözüm olarak koda alttaki iki satırı ekleyerek kodu yine standart girdi/çıkı ifadeleri kullanılarak yazılır:

```
freopen("input.txt", "r", stdin);  
freopen("output.txt", "w", stdout);
```

Bundan sonra program girdileri "input.txt" dosyasından alıp çıktıyı "output.txt" dosyasına yazdıracak.

1.3 Sayılarla Çalışmak

Tam Sayılar

Rekabetçi Programlamada en çok kullanılan tam sayı tipi `int` olup 32-bit türündedir. Bu tip $-2^{31} \dots 2^{31} - 1$ arası veya genel olarak $-2 \cdot 10^9 \dots 2 \cdot 10^9$ arası tam sayıları tutabilir. Eğer `int` istenen işlem için yeterli değilse onun yerine 64-bit `long long` kullanılabilir. `long long`, $-2^{63} \dots 2^{63} - 1$ veya genel olarak $-9 \cdot 10^{18} \dots 9 \cdot 10^{18}$ arası tam sayıları tutabilir.

Aşağıdaki kod `long long` değişkeni tanımlar:

```
long long x = 123456789123456789LL;
```

Sayının sonunda bulunan `LL`, sayının `long long` tipinde olduğunu belirtir.

`long long` tipini kullanılırken sıkça yapılan hata `int` tipinin hala kodda bir yerde kullanılmasıdır. Örneğin, aşağıdaki kod doğru gibi görünse de hata içermektedir.

```
int a = 123456789;  
long long b = a*a;  
cout << b << "\n"; // -1757895751
```

`b` değişkeni `long long` tipinde olsa da, `a*a` ifadesindeki iki sayı da `int` tipindedir ve bu işlemin sonucu olarak da `int` tipi sonuç verirler. Bu yüzden `b` yanlış sonuca sahip olacak (Bkz. Integer Overflow). Bu sorun `a` değişkeninin tipini `long long` tipine veya ifadeyi `(long long)a*a` şeklinde değiştirerek düzeltebiliriz.

Genel olarak yarışma soruları `long long` yetecek şekilde yazılır. Yine de `g++` derleyicisinin 128-bit `__int128_t` tipi sağladığını bilmekte fayda var. `__int128_t`, $-2^{127} \dots 2^{127} - 1$ arası veya genel olarak $-10^{38} \dots 10^{38}$ arası tam sayı tutabilir. Ama, bu tam sayı tipi her zaman yarışma sistemlerinde bulunmayabilir.

Modüler Aritmetik

$x \bmod m$ ifadesi, x , m 'e bölününce kalan kalanı gösterir. Örneğin, $17 \bmod 5 = 2$ olur çünkü $17 = 3 \cdot 5 + 2$.

Bazen sorunun çözümü çok büyük olduğunda, çözümü "modulo m ", vb. tipinde yazdırmamız istenebilir. (Örneğin, "modulo $10^9 + 7$ "). Burada söylenmek istenen şey, gerçek cevap çok büyük olsa dahi istenen cevap genel olarak `int` ve `long long` tipinde tutulabiliyordur.

Kalanın önemli özelliklerinden biri, kalan, toplamada, çıkarmada, çarpmada, bölmede operasyon öncesi alttaki gibi alınabilmektedir.

$$\begin{aligned}(a + b) \bmod m &= (a \bmod m + b \bmod m) \bmod m \\(a - b) \bmod m &= (a \bmod m - b \bmod m) \bmod m \\(a \cdot b) \bmod m &= (a \bmod m \cdot b \bmod m) \bmod m\end{aligned}$$

Böylece her operasyondan sonra ifadenin kalanını alarak tutulan sayının çok büyük olmaması sağlanabilir.

Örneğin aşağıdaki kod $n!$ ifadesini hesaplayacaktır., n faktöriyelinin mod m 'deki hali:

```
long long x = 1;
for (int i = 2; i <= n; i++) {
    x = (x*i)%m;
}
cout << x%m << "\n";
```

Genelde kalanın $0 \dots m - 1$ arasında olması istenir fakat C++ veya başka kodlama dillerinden kalan negatif veya sıfır olabilir. Kalanın negatif olmamasını sağlamanın yollarından biri önce kalanı hesaplayıp sonrasında sonuç negatifse m eklemektir:

```
x = x%m;
if (x < 0) x += m;
```

Fakat, bu duurm kodda çıkarma işlemi olduğu zaman negatif kalan olunca gereklidir.

Kayan Noktalı Sayılar

Genelde rekabetçi programlamadaki kayan noktalı tipler, 64-bit `double`'dir ve g++ derleyicisine ek olarak 80-bit `long double` tipi de vardır.. Çoğu durumda `double` yeterli olacaktır fakat `long double` tipi daha kesindir.

Genelde çözüm için istenen kesinlik(precision) sorunun ifadesinde verilmektedir. Çözümü yazdırmak için kolaylıkla `printf` fonksiyonunu kullanıp çıktıda bulunacak ondalık basamak sayısı belirtilebilir. Örneğin aşağıdaki kod, x değişkenini 9 tane ondalık basamak sayısıyla yazdıracaktır:

```
printf("%.9f\n", x);
```

Kayan noktalı sayılardaki zorluklardan biri, bazı sayılar kesin şekilde kayan noktalı sayı şeklinde ifade edilemeze ve bu yüzden de yuvarlama hatası oluşturalırlar. Örneğin aşağıdaki kodun sonucu ilginçtir.

```
double x = 0.3*3+0.1;
printf("%.20f\n", x); // 0.99999999999999988898
```

Yuvarlama hatasından dolayı x değişkeninin değeri 1'e eşit olması gerekirken 1'den biraz daha az olacaktır.

Kayan noktalı sayıları `==` operatörüyle karşılaştırmak risklidir çünkü karşılaştırılan değerlerin teorik olarak aynı olup pratikte kesinlik hataları olabilir. Kayan noktalı sayıları karşılaştırmamanın daha iyi bir yolu, iki sayının farkının ϵ 'dan daha küçük olduğunu kabul ederek olur. ϵ burada küçük bir sayıdır.

Pratikte sayılar aşağıdaki gibi karşılaştırılabilir: ($\epsilon = 10^{-9}$):

```
if (abs(a-b) < 1e-9) {
    // a ve b esitse
}
```

Burada dikkat edilmesi gereken şey, kayan noktalı sayıların her ne kadar tam kesin olmasa da tam sayılar (integer) belirli bir limit içerisinde kesin olarak ifade edilebilir. Örneğin `double`, mutlak değeri 2^{53} 'den küçük olan bütün tam sayıları kesin bir şekilde ifade edebilir.

1.4 Kodu Kısaltmak

Programların hızlı çalışması gerektiği için rekabetçi programlamada kısa kod idealdir. Bu yüzden rekabetçi programlamacılar data tipleri ve kodun çeşitli yerleri için daha kısa isimler tanımlarlar.

Tip İsimleri

`typedef` komutunu kullanarak bir data tipine daha kısa bir isim verilebilir.

Örneğin, isminden de anlaşılacağı üzere `long long` uzundur, bu yüzden 11 adında aşağıdaki gibi daha kısa bir isim tanımlayabiliriz:

```
typedef long long ll;
```

Bundan sonra aşağıdaki kod:

```
long long a = 123456789;
long long b = 987654321;
cout << a*b << "\n";
```

alttaki şekilde kısaltılabilir:

```
ll a = 123456789;
ll b = 987654321;
```

```
cout << a*b << "\n";
```

`typedef` komutu aynı zamanda daha karışık tipler için de kullanılabilir. Örneğin, tam sayılardan oluşan bir dinamik dizi (vector) `vi` şeklinde ve iki sayıdan oluşan bir sayı çifti ise `pi` şeklinde tanımlanabilir.

```
typedef vector<int> vi;  
typedef pair<int,int> pi;
```

Makrolar

Kodu kısaltmanın yollarından biri de **makro** (macro) tanımlamaktır. Makro, kod derlenmeden önce belirli yazıların değiştirileceğini söyler. C++'da makrolar `#define` kelimesiyle tanımlanır.

Örneğin makroları aşağıdaki gibi tanımlayabiliriz:

```
#define F first  
#define S second  
#define PB push_back  
#define MP make_pair
```

Bundan sonra kod:

```
v.push_back(make_pair(y1,x1));  
v.push_back(make_pair(y2,x2));  
int d = v[i].first+v[i].second;
```

aşağıdaki gibi kısaltılabilir:

```
v.PB(MP(y1,x1));  
v.PB(MP(y2,x2));  
int d = v[i].F+v[i].S;
```

Makro parametre de alarak döngüleri ve başka çeşitli yapıları kısaltabilir. Örneğin makro aşağıdaki gibi tanımlanabilir:

```
#define REP(i,a,b) for (int i = a; i <= b; i++)
```

Böylece kod:

```
for (int i = 1; i <= n; i++) {  
    search(i);  
}
```

aşağıdaki şekilde yazılabilir:

```
REP(i,1,n) {  
    search(i);  
}
```

Bazen makrolar fark edilmesi zor hatalar çıkartabilir. Örneğin, aşağıdaki makronun, sayının karesini hesapladığını varsayalım:

```
#define SQ(a) a*a
```

Bu makro her zaman beklendiği gibi *ÇALIŞMAYABİLİR*. Örneğin bu kod

```
cout << SQ(3+3) << "\n";
```

aşağıdaki koda denktir:

```
cout << 3+3*3+3 << "\n"; // 15
```

Bu makro aşağıdaki gibi daha iyi bir şekilde yazılabilir:

```
#define SQ(a) (a)*(a)
```

Şimdi bu kod

```
cout << SQ(3+3) << "\n";
```

aşağıdaki koda denktir:

```
cout << (3+3)*(3+3) << "\n"; // 36
```

1.5 Matematik

Matematik rekabetçi programlamada önemli bir rol oynar ve güçlü matematik yeteneğine sahip olmayan başarılı bir rekabetçi programlamacı olamaz. Bu bölüm, kitapta gerekli olan önemli matematik konseptlerinden ve formüllerinden bahsediyor.

Toplum Formülleri

$$\sum_{x=1}^n x^k = 1^k + 2^k + 3^k + \dots + n^k,$$

formundaki her toplamın, $k + 1$ derecesinde bir polinom olan bir kapalı form formülü vardır. Örneğin ²

$$\sum_{x=1}^n x = 1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2}$$

² Bu tip toplamlar için **Faulhaber'in Formülü** adında genel bir formül vardır fakat bu kitap için çok karmaşıktır.

ve

$$\sum_{x=1}^n x^2 = 1^2 + 2^2 + 3^2 + \dots + n^2 = \frac{n(n+1)(2n+1)}{6}.$$

Bir **aritmetik ilerleme** (arithmetic progression) her iki ardışık elemanın arasındaki farkın sabit olduğu sayılar dizisidir. Örneğin

$$3, 7, 11, 15$$

dizisi 4 sabitin olduğu aritmetik ilerleme örneğidir. Aritmetik ilerlemenin toplamı

$$\underbrace{a + \dots + b}_{n \text{ numbers}} = \frac{n(a+b)}{2}$$

formülü ile hesaplanır ki burada a ilk sayı, b son sayı ve n de sayı miktarını verir. Örneğin

$$3 + 7 + 11 + 15 = \frac{4 \cdot (3 + 15)}{2} = 36.$$

Bu formül, toplamın n tane sayıdan oluşan ve her sayının değerinin ortalamada $(a+b)/2$ olmasından gelir.

Geometrik ilerleme (geometric progression), her iki ardışık sayı arasındaki oranın sabit olduğu sayılar dizisidir. Örneğin

$$3, 6, 12, 24$$

sabitin 2 olduğu bir geometrik ilerlemedir. Geometrik ilerleme toplamı a 'nın ilk sayı olduğu b 'nin son sayı olduğu ve ardışık sayılar arasındaki oranın k olduğu

$$a + ak + ak^2 + \dots + b = \frac{bk - a}{k - 1}$$

formülü ile hesaplanabilir. Örneğin

$$3 + 6 + 12 + 24 = \frac{24 \cdot 2 - 3}{2 - 1} = 45.$$

Formül aşağıdaki gibi hesaplanabilir. Şimdi

$$S = a + ak + ak^2 + \dots + b$$

olsun. İki tarafı da k ile çarparak

$$kS = ak + ak^2 + ak^3 + \dots + bk$$

ifadesini elde ederiz ve

$$kS - S = bk - a$$

eşitliği çözülerek formül verilir.

Geometrik ilerlemenin toplamının özel bir durumu

$$1 + 2 + 4 + 8 + \dots + 2^{n-1} = 2^n - 1.$$

Harmonik toplam,

$$\sum_{x=1}^n \frac{1}{x} = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}$$

formunda olan toplamdır.

Harmonik toplamın üst sınırı $\log_2(n)+1$. Spesifik olarak her $1/k$ terimini öyle bir değiştiririz ki k , k 'yi geçmeyen en büyük ikinin kuvvet iolur. Örneğin $n = 6$ ise toplamı aşağıdaki tahmin edebiliriz:

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} \leq 1 + \frac{1}{2} + \frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4}.$$

Bu üst sınır $\log_2(n)+1$ parçadan oluşur ($1, 2 \cdot 1/2, 4 \cdot 1/4$, vb.) ve her parçanın değeri en fazla 1'dir.

Küme Teorisi

Bir **küme**, elemanların grubudur. Örneğin

$$X = \{2, 4, 7\}$$

kümesi 2, 4 ve 7 elemanlarını içerir. \emptyset sembolü boş kümeyi gösterir. $|S|$, S kümesinin büyüklüğünü yani kümede kaç eleman bulunduğunu gösterir. Örneğin yukarıdaki kümede $|X| = 3$.

Eğer S kümesi x elemanını içeriyorsa $x \in S$ yazarız ve eğer yoksa $x \notin S$ yazarız. Örneğin yukarıdaki kümede

$$4 \in X \text{!} \quad \text{vex!} \quad 5 \notin X.$$

Yeni kümeler, küme operasyonlarıyla oluşturulabilir.

- **Kesişim** $A \cap B$, hem A hem de B 'de bulunan elemanlardan oluşur. Örneğin eğer $A = \{1, 2, 5\}$ ve $B = \{2, 4\}$ ise $A \cap B = \{2\}$.
- **Birleşim** $A \cup B$ ya A ya B ya da ikisinde de bulunan elemanlardan oluşur. Örneğin eğer $A = \{3, 7\}$ ve $B = \{2, 3, 8\}$ ise $A \cup B = \{2, 3, 7, 8\}$.
- **Tamlayan** \bar{A} A 'da bulunmayan elemanlardan oluşur. Bir tamlayanın genişliği **evrensel kümeye** yani bütün olası elemanları içeren kümeye bağlıdır. Örneğin eğer $A = \{1, 2, 5, 7\}$ ve evrensel küme $\{1, 2, \dots, 10\}$ ise $\bar{A} = \{3, 4, 6, 8, 9, 10\}$.
- **Fark** $A \setminus B = A \cap \bar{B}$, A 'da bulunup B 'de bulunmayan elemanlardan oluşur. Bu arada B , A 'da bulunmayan elemanlar içerebilir. Örneğin eğer $A = \{2, 3, 7, 8\}$ ve $B = \{3, 5, 8\}$ ise $A \setminus B = \{2, 7\}$.

Eğer A 'daki her eleman aynı zamanda S 'ye de aitse A , S 'nin **altkümesi** deriz ki bunu da $A \subset S$ ile gösteririz. Bir S kümesi her zaman (boş küme de dahil olmak üzere) $2^{|S|}$ altküme içerir. Örneğin $\{2, 4, 7\}$ kümesinin alt kümeleri

$\emptyset, \{2\}, \{4\}, \{7\}, \{2, 4\}, \{2, 7\}, \{4, 7\}$ ve $\{2, 4, 7\}$.

Bazı sık kullanılan kümeler \mathbb{N} (doğal sayılar), \mathbb{Z} (tamsayılar), \mathbb{Q} (rasyonel sayılar) and \mathbb{R} (reel sayılar). \mathbb{N} kümesi duruma göre iki şekilde tanımlanabilir: ya $\mathbb{N} = \{0, 1, 2, \dots\}$ ya da $\mathbb{N} = \{1, 2, 3, \dots\}$.

Aynı zamanda $f(n)$ 'in bir fonksiyon olduğu

$$\{f(n) : n \in S\},$$

formundaki bir kuralı kullanarak küme oluşturabiliriz. Bu küme, n 'in S 'teki olduğu her eleman için $f(n)$ formundaki bütün elemanları bulundurur. Örneğin

$$X = \{2n : n \in \mathbb{Z}\}$$

kümesi bütün çift sayıları içerir.

Mantık

Bir mantık gösteriminin değeri ya **doğru** (1) ya da **yanlış** (0) olabilir. En önemli mantık operasyonları \neg (**değil**), \wedge (**ve**), \vee (**veya**), \Rightarrow (**ise**) ve \Leftrightarrow (**ancak ve ancak**). Aşağıdaki tablo bu operasyonların anlamlarını gösterir:

A	B	$\neg A$	$\neg B$	$A \wedge B$	$A \vee B$	$A \Rightarrow B$	$A \Leftrightarrow B$
0	0	1	1	0	0	1	1
0	1	1	0	0	1	1	0
1	0	0	1	0	1	0	0
1	1	0	0	1	1	1	1

$\neg A$ gösterimi A 'nın tersi değerine sahiptir. $A \wedge B$ gösterimi de eğer hem A hem B doğruysa doğrudur. $A \vee B$ gösterimi ya A ya B ya da her ikisi de doğruysa doğrudur. $A \Rightarrow B$ gösterimi, A doğru olduğu zaman B doğru ise doğrudur. $A \Leftrightarrow B$ gösterimi eğer hem A hem de B ikisi de doğruysa ya da ikisi de yanlışsa doğrudur.

Predicate, parametreye bağlı olarak doğru ya da yanlış olan bir gösterimdir. Predicatelar büyük harflerle gösterilir. Örneğin $P(x)$, x asal sayı olursa doğru olacak şekilde tanımlayabiliriz. Bu tanımlı kullanarak $P(7)$ doğru olur ama $P(8)$ yanlış olur. is an expression that is true or false

Niceleyici bir kümenin elemanlarına bir mantık operasyonu bağlar. En önemli niceleyiciler \forall (**her**) ve \exists (**bazı**) olur. Örneğin

$$\forall x(\exists y(y < x))$$

kümedeki her x elemanı için öyle bir y elemanı vardır ki y , x 'ten daha küçüktür. Bu tamsayılar için doğrudur fakat doğal sayılar için yanlıştır.

Yukarıdaki gösterimi kullanarak çeşitli mantık gösterimlerini gösterebiliriz. Örneğin

$$\forall x((x > 1 \wedge \neg P(x)) \Rightarrow (\exists a(\exists b(a > 1 \wedge b > 1 \wedge x = ab))))$$

yani eğer bir x sayısı 1'den büyükse ve asal sayı değilse 1'den büyük olup çarpımları x olan a ve b sayıları vardır. Bu gösterim tamsayılar kümesi için doğrudur.

Fonksiyonlar

$\lfloor x \rfloor$ fonksiyonu x sayısını en yakın küçün tam sayıya yuvarlarken $\lceil x \rceil$ x sayısını en yakın büyük tam sayıya yuvarlar. Örneğin

$$\lfloor 3/2 \rfloor = 1x! \quad \text{veya} \quad \lceil 3/2 \rceil = 2.$$

$\min(x_1, x_2, \dots, x_n)$ ve $\max(x_1, x_2, \dots, x_n)$ fonksiyonları x_1, x_2, \dots, x_n değerlerinin en büyük ve en küçük değerlerini verir. Örneğin

$$\min(1, 2, 3) = 1x! \quad \text{veya} \quad \max(1, 2, 3) = 3.$$

Faktöriyel $n!$

$$\prod_{x=1}^n x = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n$$

olarak ya da özyinelemeli olarak

$$\begin{aligned} 0! &= 1 \\ n! &= n \cdot (n-1)! \end{aligned}$$

şeklinde tanımlanabilir.

Fibonacci sayıları çok durumda ortaya çıkar. Bu sayılar özyinelemeli bir şekilde tanımlanabilir:

$$\begin{aligned} f(0) &= 0 \\ f(1) &= 1 \\ f(n) &= f(n-1) + f(n-2) \end{aligned}$$

İlk Fibonacci sayıları

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \dots$$

Aynı zamanda Fibonacci sayılarını hesaplamak için kapalı formda bir formül vardır ki bu aynı zamanda **Binet'in formülü** olarak da söylenir:

$$f(n) = \frac{(1 + \sqrt{5})^n - (1 - \sqrt{5})^n}{2^n \sqrt{5}}.$$

Logaritmalar

Bir x sayısının **logaritması** $\log_k(x)$ olarak gösterilir ve burada k logaritmanın tabanıdır. Tanıma göre $\log_k(x) = a$ tam $k^a = x$ olduğu zaman olur.

Logaritmanın işe yarar bir özelliği $\log_k(x)$ 'in, x 'i 1 sayısına getirmek için kaç defa k 'e böleceğimize eşit olmasıdır. Örneğin $\log_2(32) = 5$ çünkü 5 defa 2'ye bölmemiz gerekir:

$$32 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$$

Logaritmalar genelde algoritmanın analizinde kullanılır çünkü çoğu verimli algoritma her adımda bir şeyi yarıya böler. Bu yüzden algoritmaların verimliliklerini logaritma ile tahmin edebiliriz.

Bir çarpımın logaritması

$$\log_k(ab) = \log_k(a) + \log_k(b)$$

ve sonuç olarak

$$\log_k(x^n) = n \cdot \log_k(x).$$

Bunla beraber bir bölümün logaritması

$$\log_k\left(\frac{a}{b}\right) = \log_k(a) - \log_k(b).$$

Başka işe yarar bir formül ise

$$\log_u(x) = \frac{\log_k(x)}{\log_k(u)}$$

ve bunu kullanarak eğer sabit bir tabanda logaritmayı hesaplamamanın bir yolu varsa herhangi bir tabanın logaritmasını hesaplamak mümkündür

Bir x sayısının **doğal logaritması** $\ln(x)$, tabanı $e \approx 2.71828$ olan bir logaritmadır. Logaritmaların başka bir özelliği ise b tabanındaki bir x sayısının basamak sayısı $\lfloor \log_b(x) + 1 \rfloor$. Örneğin 2 tabanındaki 123 sayısının gösterimi 1111011 olur ve $\lfloor \log_2(123) + 1 \rfloor = 7$.

1.6 Yarışmalar ve Kaynaklar

IOI

International Olympiad in Informatics (IOI) (Uluslararası Bilgisayar Olimpiyatı) yıllık olarak lise öğrencilerinin katıldığı yarışmadır. Her ülke yarışmaya 4 kişilik bir takım gönderebilir. Genel olarak 80 ülkeden 300 katılımcı katılır.

IOI, 2 tane 5 saatlik yarışmadan oluşur. İki yarışmada da yarışmacıların farklı zorluğa sahip 3 algoritma problemi çözmesi istenir. Problemler alt görevlerden oluşur ve her birinin belirli miktarda puanı vardır. Yarışmacılar takımlar halinde gelmiş olsalar da bireysel olarak yarışır.

IOI müfredatı [41] IOI’da çıkabilecek konuları içerir. Neredeyse IOI müfredatındaki her konu bu kitapta anlatıldı.

IOI’ya katılacak olan yarışmalar ulusal yarışmalarla belirlenir (Örneğin Türkiye’de TÜBİTAK Bilim Olimpiyatları). IOI’dan önce çeşitli bölgeler yarışmalar düzenlenir. Bunlardan bazıları Baltic Olympiad in Informatics (BOI) (Baltık Bilgisayar Olimpiyatı), Central European Olympiad in Informatics (CEOI) (Merkez Avrupa Bilgisayar Olimpiyatı) ve Asia-Pacific Informatics Olympiad (APIO) (Asya Pasifik Bilgisayar Olimpiyatı) idir.

Bazı ülkeler gelecek IOI katılımcıları için online antrenman yarışmaları düzenler. Mesala Hırvatistan, Croatian Open Competition in Informatics [11] (Hırvat Açık Bilgisayar Yarışması) yarışmasını ve Amerika da USA Computing Olympiad (Amerika Bilgisayar Olimpiyatı) yarışmasını [68] düzenler. Bunla beraber bir sürü Polonya yarışmasının soruları [60] internette bulunabilir.

ICPC

The International Collegiate Programming Contest (ICPC) (Uluslararası Üniversite Programlama Yarışması) üniversite öğrencileri için yıllık düzenlenen bir programlama yarışmasıdır. Her takım üç öğrenciden oluşur ve IOI'n aksine öğrenciler birlikte çalışır ve her takım için bir bilgisayar vardır.

ICPC birkaç aşamadan oluşur ve en sonda en iyi takımlar dünya finallerine çağrılır. Yarışmalarda on binlerce yarışmacı olsa da final yarışmasına çok kısıtlı sayıda³ yer vardır yani final yarışmasına bile gelebilmek bazı bölgeler için büyük bir başarıdır.

Her ICPC yarışmasında takımlar 10 algoritma sorusunu 5 saatte çözmeye çalışır. Bir problemin çözümü bütün test durumlarını verimli bir şekilde çözerse doğru kabul edilir. Yarışma sırasında yarışmacılar diğer takımların sonuçlarına bakabilirler ama son saatte sıralama dondurulur ve son gönderilerin sonuçları görünemez.

ICPC'de çıkabilen konular IOI'daki gibi çok belirli değildir. Yine de ICPC'de daha fazla bilgi gerekir ki özellikle matematik alanında gerekir.

Online Yarışmalar

Bu arada herkese açık olan online yarışmalar da vardır. Şu anda en aktif yarışma sitesi Codeforces'dur ki genelde haftalık yarışmalar düzenlenir. Codeforces'da yarışmacılar iki kategoriye ayrılmıştır. Yeni yarışmacılar Div2'de kapışırken daha deneyimli yarışmacılar Div1'de yarışır. Bunlara ekstradan bazen Div3 ve Div4 yarışmaları da yapılmaktadır. Diğer yarışma siteleri AtCoder, CS Academy, HackerRank ve Topcoder idir.

Bazı şirketlerde final elemeleri için online yarışmalar düzenlerler. Bunlardan bazıları Facebook Hacker Cup, Google Code Jam ve Yandex.Algorithm'dir. Tabii ki şirketler, yarışmaları çalışan bulmak için de kullanır çünkü bir yarışmada iyi performans sergilemek o kişinin yeteneklerini gösterir.

Kitaplar

Bu kitap haricinde rekabetçi programlama ve algoritmik problem çözümü için bazı kitaplar vardır fakat bu kitapların Türkçe çevirisi bulunmama ihtimali vardır:

- S. S. Skiena and M. A. Revilla: *Programming Challenges: The Programming Contest Training Manual* [59]
- S. Halim and F. Halim: *Competitive Programming 3: The New Lower Bound of Programming Contests* [33]
- K. Diks et al.: *Looking for a Challenge? The Ultimate Problem Set from the University of Warsaw Programming Competitions* [15]

³Final yarışmasına yer sayısı yıldan yıla göre değişir. Mesela 2017'de 133 takımlik yer vardı.

İlk iki kitap başlangıç seviyesi için yapıldıysa da son kitap daha gelişmiş konuları içerir.

Tabii rekabetçi programlamacılar için genel algoritma kitapları da uygundur. Bazı popüler kitaplar:

- T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein: *Introduction to Algorithms* [13]
- J. Kleinberg and É. Tardos: *Algorithm Design* [45]
- S. S. Skiena: *The Algorithm Design Manual* [58]

Bölüm 2

Zaman Karmaşıklığı

Rekabetçi programlamada algoritmaların verimliliği önemlidir. Genelde soruyu çözen yavaş bir algoritma oluşturmak kolaydır ama sorun asıl zorluğu hızlı bir algoritma oluşturmaktır. Eğer algoritma çok yavaşsa ya sorudan kısmi puan alacaktır ya da hiç almayacaktır.

Zaman karmaşıklığı bir algoritmanın bir girdi için tahmini olarak ne kadar süre gerektireceğini verir. Bunun amacı, girdinin boyutuna göre değişen ve algoritmanın verimliliğini gösteren bir fonksiyon oluşturmaktır. Zaman karmaşıklığını hesaplayarak algoritmayı koda dökmeden önce algoritmanın yeterince hızlı olup olmadığını bulmuş oluruz.

2.1 Hesaplama Kuralları

Bir algoritmanın zaman karmaşıklığı $O(\dots)$ ile gösterilir. Burada üç nokta, herhangi bir fonksiyonu belirtir. Genelde, girdi büyüklüğü olarak n değişkeni kullanılır. Örneğin sayılardan oluşan bir dizide n dizinin büyüklüğü ve eğer girdi bir yazı ise n yazının uzunluğunu verir.

Döngüler

Algoritmanın yavaş çalışmasına sebep olan genel sebeplerden biri de girdi üzerinde çalışan çok fazla döngü bulundurmasıdır. Bir algoritma ne kadar çok iç içe geçmiş döngü bulundurursa o kadar yavaşlar. Örneğin eğer k tane iç içe geçmiş döngü varsa zaman karmaşıklığı $O(n^k)$ olur.

Örneğin aşağıdaki kodun zaman karmaşıklığı $O(n)$ 'dir:

```
for (int i = 1; i <= n; i++) {  
    // kod  
}
```

Ve aşağıdaki kodun zaman karmaşıklığı ise $O(n^2)$ 'dir:

```
for (int i = 1; i <= n; i++) {  
    for (int j = 1; j <= n; j++) {  
        // kod  
    }  
}
```

```
}  
}
```

Büyüklik Sırası (Order of Magnitude)

Zaman karmaşıklığı bize döngü içindeki kodun tam olarak ne kadar defa çalışacağını vermez. Örneğin aşağıdaki örneklerde döngü içindeki kod $3n$, $n+5$ ve $\lfloor n/2 \rfloor$ defa çalışmıştır ama hepsinde kodların zaman karmaşıklığı $O(n)$ 'dir.

```
for (int i = 1; i <= 3*n; i++) {  
    // kod  
}
```

```
for (int i = 1; i <= n+5; i++) {  
    // kod  
}
```

```
for (int i = 1; i <= n; i += 2) {  
    // code  
}
```

Örneğin aşağıdaki kodun zaman karmaşıklığı $O(n^2)$ 'dir:

```
for (int i = 1; i <= n; i++) {  
    for (int j = i+1; j <= n; j++) {  
        // kod  
    }  
}
```

Aşamalar (Phases)

Eğer algoritma birden çok aşamadan oluşuyorsa toplam zaman karmaşıklığı, en büyük zaman karmaşıklığına sahip aşamanın zaman karmaşıklığına eşittir. Böyle yapılmasının nedeni ise genelde en yavaş çalışan aşamanın kodu yavaşlatan aşama olmasıdır.

Örneğine aşağıdaki kod $O(n)$, $O(n^2)$ ve $O(n)$ zaman karmaşıklığına sahip üç tane aşamadan oluşmaktadır. Bu yüzden toplam zaman karmaşıklığı $O(n^2)$ 'e eşittir.

```
for (int i = 1; i <= n; i++) {  
    // kod  
}  
for (int i = 1; i <= n; i++) {  
    for (int j = 1; j <= n; j++) {  
        // kod  
    }  
}
```



```

    }
}
for (int i = 1; i <= n; i++) {
    // code
}

```

Çoklu Değişkenler

Bazen zaman karmaşıklığı birkaç faktöre bağlıdır. Bu durumda zaman karmaşıklığı formülü birden çok değişken içerir.

Örneğin aşağıdaki kodun zaman karmaşıklığı $O(nm)$ 'dir:

```

for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
        // kod
    }
}

```

Özyineleme(Recursion)

Özyinelemeli bir fonksiyonun zaman karmaşıklığı, fonksiyonun ne kadar çağrıldığına ve bir çağrının zaman karmaşıklığına bağlıdır. Toplam zaman karmaşıklığı bu değerlerin çarpımına eşittir.

Örneğin aşağıdaki fonksiyonda:

```

void f(int n) {
    if (n == 1) return;
    f(n-1);
}

```

$f(n)$, n defa fonksiyonu çağırır ve her çağrının zaman karmaşıklığı $O(1)$ 'e eşittir. Böylece toplam zaman karmaşıklığı $O(n)$ 'e eşittir.

Başka örnek olarak, aşağıdaki fonksiyonda:

```

void g(int n) {
    if (n == 1) return;
    g(n-1);
    g(n-1);
}

```

Her fonksiyon $n = 1$ haricinde iki defa çağrı oluşturur. Mesela g fonksiyonunu n parametresiyle çağırdığımız zamanki karmaşıklığa bakalım. Aşağıdaki tablo tek çağrıdan oluşan toplam fonksiyon çağrı sayısını gösterir.

fonksiyon çağrısı	toplam çağrı
$g(n)$	1
$g(n-1)$	2
$g(n-2)$	4
...	...
$g(1)$	2^{n-1}

Buna göre toplam zaman karmaşıklığı:

$$1 + 2 + 4 + \dots + 2^{n-1} = 2^n - 1 = O(2^n).$$

2.2 Karmaşıklık Sınıfları (Complexity classes)

Aşağıdaki liste yaygın olan algoritma zaman karmaşıklıklarını gösteriyor:

$O(1)$ Sabit zamanlı (Constant Time) algoritmaların zaman karmaşıklığı girdi büyüklüğüne bağlı değildir. Tipik bir sabit zaman algoritması direkt olarak cevabı hesaplayan bir formüldür.

$O(\log n)$ Logaritmik algoritma genelde girdi büyüklüğünü her adımda ikiye böler. Bu tip bir algoritmanın zaman karmaşıklığı logaritmiktir çünkü n 'i 1'e getirmek için gereken 2'ye bölme sayısı $\log_2 n$ 'ye eşittir.

$O(\sqrt{n})$ Karakök algoritması $O(\log n)$ 'den yavaştır ama $O(n)$ 'den hızlıdır. Karaköklere bir özelliği $\sqrt{n} = n/\sqrt{n}$ idir yani karakök e square \sqrt{n} girdinin ortasında durur.

$O(n)$ Linear algoritma girdiden sabit bir sayıda geçer. Bu genelde olabilen en iyi zaman karmaşıklığıdır çünkü çoğunlukla cevabı bulmak için her elemana en az bir defa bakmak gerekir.

$O(n \log n)$ Bu zaman karmaşıklığı genelde algoritmanın girdiyi sıraladığını gösterir çünkü verimli bir sıralama algoritmasının zaman karmaşıklığı $O(n \log n)$ 'dir. Başka bir olasılık ise algoritmanın her işlemi $O(\log n)$ tutan bir data yapısı kullanmasıdır..

$O(n^2)$ Quadratic algoritma genelde iki tane iç içe döngü bulundurur. Böylece girdideki bütün çiftlerden geçmek $O(n^2)$ zamanda mümkündür.

$O(n^3)$ Kübik (cubic) algoritma genelde üç tane iç içe döngü bulundurur. Böylece girdideki bütün üçlü elemanları $O(n^3)$ zamanda dolaşmak mümkündür.

$O(2^n)$ Bu zaman karmaşıklığı genelde algoritmanın girdideki bütün alt kümeleri dolaştığını gösterir. Örneğin $\{1, 2, 3\}$ dizisinin alt kümeleri \emptyset , $\{1\}$, $\{2\}$, $\{3\}$, $\{1, 2\}$, $\{1, 3\}$, $\{2, 3\}$ ve $\{1, 2, 3\}$ 'dir.

$O(n!)$ Bu zaman karmaşıklığı genelde algoritmanın girdideki bütün elemanların permütasyonlarını dolaştığını gösterir.

Örneğin $\{1, 2, 3\}$ dizisinin permütasyonları $(1, 2, 3)$, $(1, 3, 2)$, $(2, 1, 3)$, $(2, 3, 1)$, $(3, 1, 2)$ ve $(3, 2, 1)$ 'dir.

Bir algoritmanın zaman karmaşıklığı eğer en fazla $O(n^k)$ ise, k sabit sayı, bu algoritma bir *polinomdur*. Yukarıdaki bulunan zaman karmaşıklıklarından $O(2^n)$ ve $O(n!)$ hariç diğer hepsi polinomdur. Pratikte k sabiti genelde küçüktür ve bu yüzden genelde polinom zaman karmaşıklığı *verimlidir*.

Bu kitaptaki algoritmaların çoğu polinomdur. Yine de polinom algoritma çözümü olmayan bir sürü önemli problem vardır. Başka anlatımla kimse bu problemleri verimli şekilde nasıl çözeceğini bilmemektedir. **NP-hard** problemler, bu tip problemlerin önemli parçasıdır ve bunlar için herhangi bir polinom algoritma bulunmamaktadır.¹

2.3 Verimliliği Tahmin Etme

Algoritmanın zaman karmaşıklığı hesaplanarak algoritmayı koda dökmeden önce yeterince verimli olup olmadığı kontrol edilebilir. Tahminlerin başlangıç noktası modern bir bilgisayarın saniyede yüz milyonlarca işlem yapabilmesinde yatar.

Örneğin girdi boyutu $n = 10^5$ olan ve zaman limiti 1 saniye olan bir problem düşünün. Eğer çözümün zaman karmaşıklığı $O(n^2)$ ise, algoritma ortalama $(10^5)^2 = 10^{10}$ işlem uygular. Bunun için de en az onlarca saniye gerekir yani bu algoritma bu soruyu çözmek için çok yavaş kalır.

Diğer bir yandan, girdinin büyüklüğü verildiği varsayılırsa soruyu çözmek için gerekli olan algoritmanın zaman karmaşıklığını *tahmin edebiliriz*. Aşağıdaki tablo, zaman limitinin 1 saniye olduğu durumda işe yarar tahminleri göstermektedir:

Girdi Büyüklüğü	Gerekli Zaman Karmaşıklığı
$n \leq 10$	$O(n!)$
$n \leq 20$	$O(2^n)$
$n \leq 500$	$O(n^3)$
$n \leq 5000$	$O(n^2)$
$n \leq 10^6$	$O(n \log n)$ veya $O(n)$
n büyük	$O(1)$ veya $O(\log n)$

Örneğin eğer girdinin büyüklüğü $n = 10^5$ ise büyük ihtimalle beklenen zaman karmaşıklığı ya $O(n)$ 'dir ya da $O(n \log n)$ 'dir. Bu bilgi algoritmayı bulmayı kolaylaştırır çünkü bu yöntem daha kötü zaman karmaşıklığına sahip algoritmaları kaldırır.

Yine de, zaman karmaşıklığının sadece verimliliği tahmin eden bir şey olduğunu hatırlamakta fayda var çünkü zaman karmaşıklığı *sabit faktörleri* saklar. Örneğin $O(n)$ zamanda çalışan bir algoritma $n/2$ veya n işlem yapabilir. Bu durum, algoritmanın gerçek çalışma süresini ciddi şekilde etkiler.

2.4 En Büyük Alt dizi Toplamı

Genelde bir soruyu çözebilen algoritmalar farklı zaman karmaşıklığına sahiptir. Bu bölüm düz $O(n^3)$ çözüme sahip klasik bir problemi anlatacak. Fakat daha

¹Bu konu hakkında klasik kitaplardan biri M. R. Garey's and D. S. Johnson's *Computers and Intractability: A Guide to the Theory of NP-Completeness* [28].

iyi algoritmalar oluşturularak bu soruyu $O(n^2)$ zamanda ve hatta $O(n)$ zamanda çözmek mümkündür.

Soruda istenen, n elemanlı bir dizideki en büyük alt dizi toplamını bulmaktır yani ardışık eleman sıralarından en büyük toplama sahip sırayı bulmaktır.² Bu soru, dizide negatif değerler bulunabileceğinden dolayı ilginçtir.

Örneğin aşağıdaki dizide:

-1	2	4	-3	5	2	-5	2
----	---	---	----	---	---	----	---

aşağıdaki alt dizi en büyük toplam olan 10'u vermektedir.:

-1	2	4	-3	5	2	-5	2
----	---	---	----	---	---	----	---

Soruda boş alt dizinin de alınabileceğini kabul ettik. Böylece maksimum alt dizi toplamı en az 0'dır.

1. Algoritma

Düz çözüm, bütün alt dizileri denemektir. Böylece her alt dizinin değerlerinin toplamını bularak en büyük toplamı bulabiliriz. Aşağıdaki kod bu anlatılan algoritmayı uygular:

```
int best = 0;
for (int a = 0; a < n; a++) {
    for (int b = a; b < n; b++) {
        int sum = 0;
        for (int k = a; k <= b; k++) {
            sum += array[k];
        }
        best = max(best, sum);
    }
}
cout << best << "\n";
```

a ve b değişkenleri alt dizinin ilk ve son indisini sabitler. sum değişkeni ile alt dizi toplamı hesaplanır. $best$ değişkeni de arama sırasındaki en büyük toplamı tutar.

Bu algoritmanın zaman karmaşıklığı $O(n^3)$ 'tür çünkü girdiyi dolaşmak için üç tane iç içe döngü bulunmaktadır.

2. Algoritma

1. Algoritmayı, sadece bir tane döngü çıkararak daha verimli hale getirmek mümkün. Bunu alt dizinin sonuna doğru ilerlerken aynı zamanda toplamı hesaplayarak bulmak mümkün.

²J. Bentley'nin *Programming Pearls* [8] kitabı bu soruyu ünlü yapmıştır.

```

int best = 0;
for (int a = 0; a < n; a++) {
    int sum = 0;
    for (int b = a; b < n; b++) {
        sum += array[b];
        best = max(best, sum);
    }
}
cout << best << "\n";

```

Böylece zaman karmaşıklığı $O(n^2)$ oluyor.

3. Algoritma

İlginç bir şekilde soruyu $O(n)$ zamanda yani tek bir döngüyle çözmek mümkün.³

Çözüm fikri, her dizi pozisyonu için o pozisyonda biten maksimum alt dizi toplamını bulmaktır. Bundan sonra sonuç, bu toplamardan maksimum olandır.

Başka bir soru olarak k . pozisyonda biten en büyük alt dizi toplamını bulmamız isteniyor. İki seçenek var:

1. Alt dizi sadece k . indiste bulunan elemanı kapsıyor.
2. Alt dizi $k - 1$. pozisyonda biten bir alt diziden oluşuyor ve ekstradan k . pozisyonda biten elemanı kapsıyor.

İkinci durumda, en büyük toplama sahip alt diziyi bulmak istediğimiz için $k - 1$. pozisyonda biten alt dizinin de ayrıca en büyük toplama sahip olması gerekir. Böylece bu soruyu verimli bir şekilde her bitiş pozisyonunun maksimum alt dizi toplamını soldan sağa hesaplayıp bulabiliriz.

Aşağıdaki kod belirtilen algoritmayı uygular:

```

int best = 0, sum = 0;
for (int k = 0; k < n; k++) {
    sum = max(array[k], sum+array[k]);
    best = max(best, sum);
}
cout << best << "\n";

```

Algoritma girdiden sadece bir döngü çalıştırır ve böylece zaman karmaşıklığı $O(n)$ olur. Bu aynı zamanda bu soru için en iyi zaman karmaşıklığıdır çünkü soru için her algoritma bütün dizi elemanlarından en az bir defa geçmesi gerekir.

Verimlilik Karşılaştırması

Gerçekte bu algoritmaların ne kadar verimli olduğunu incelemek ilginçtir. Aşağıdaki tablo yukarıdaki algoritmaların modern bir bilgisayarda farklı n değerleri

³In [8], bu linear zamanlı algoritma J. B. Kadane'a aittir ve bazen bu algoritma **Kadane'in algoritması** olarak isimlendirilir.

için çalışma zamanlarını gösteriyor. Her testte, girdi karışık bir şekilde oluşturulmuştur. Girdileri okumak için gereken zaman ölçülmemiştir.

Dizi Büyüklüğü n	1. Algoritma	2. Algoritma	3. Algoritma
10^2	0.0 s	0.0 s	0.0 s
10^3	0.1 s	0.0 s	0.0 s
10^4	> 10.0 s	0.1 s	0.0 s
10^5	> 10.0 s	5.3 s	0.0 s
10^6	> 10.0 s	> 10.0 s	0.0 s
10^7	> 10.0 s	> 10.0 s	0.0 s

Buradaki karşılaştırma, girdi büyüklüğü küçük olduğu zaman bütün algoritmaların verimli olduğunu gösteriyor ama daha büyük girdilerde algoritmaların çalışma süreleri arasındaki farklar belirginleşiyor. 1. Algoritma $n = 10^4$ olduğu zaman ve 2. Algoritma $n = 10^5$ olduğu zaman yavaş kalıyor. Sadece 3. Algoritma, en büyük girdileri yeterince hızlı bir şekilde işleyebiliyor.

Bölüm 3

Sıralama

Sıralama temel algoritma sorularından biridir. Çoğu algoritmanın içeriğinde veriyi sıralı halde işlemek daha kolay olduğu için sıralama bulunur.

Örneğin "Bu dizide birbiriyle aynı iki eleman var mı?" sorusu sıralamayla çok kolay bir şekilde yapılabilir. Eğer dizi birbiriyle aynı iki eleman içeriyorsa dizi sıralandıktan sonra bunlar ardışık olacaktır. Böylece kolay bir şekilde bulunabilir. Aynı şekilde "Bu dizide en fazla bulunan eleman hangisidir?" sorusu da benzer şekilde çözülebilir.

Çeşitli sıralama algoritmaları bulunur ve bunlar nasıl farklı algoritma tekniklerinin uygulanabileceğini gösteren örneklerdir. Verimli çalışan sıralama algoritmaları $O(n \log n)$ zamanda çalışır ve genelde içeriğinde sıralama bulunan algoritmalar da bu zaman karışıklığına (time complexity) sahiptir.

3.1 Sıralama Teorisi

Temel sıralama problemi şu şekildedir:

n elemana sahip bir diziyi artan sırada sıralayın.

Örneğin aşağıdaki dizi:

1	3	8	2	9	2	5	6
---	---	---	---	---	---	---	---

sıralandıktan sonra aşağıdaki hale dönüşür:

1	2	2	3	5	6	8	9
---	---	---	---	---	---	---	---

$O(n^2)$ algoritmalar

Basit dizi sıralama algoritmaları $O(n^2)$ zamanda çalışır. Bu algoritmalar kısa olup genelde iki for döngüsüyle çalışır. Çok bilinen $O(n^2)$ algoritmalarından biri olan **kabarcık sıralaması (bubble sort)**, dizideki elemanların değerlerine göre "kabarcık" şeklinde olmalarına göre yapılır.

Kabarcık sıralamasında n defa tur dönülür. Her turda algoritma, dizideki elemanları tarar. Algoritma sıraya uymayan iki ardışık eleman bulduğu zaman birbirlerinin yerleri değiştirilir. Kabarcık algoritması aşağıdaki şekilde koda dökülebilir:

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n-1; j++) {
        if (array[j] > array[j+1]) {
            swap(array[j], array[j+1]);
        }
    }
}
```

İlk tur bittiği zaman en büyük eleman doğru pozisyonda bulunur. Genel olarak k tur sonra, en büyük k eleman doğru sırada olacaktır. Bu yüzden n tur sonra bütün dizi sıralanmış olacaktır.

Örneğin aşağıdaki dizi:

1	3	8	2	9	2	5	6
---	---	---	---	---	---	---	---

kabarcık sıralamasının ilk turunda elemanlar aşağıdaki gibi yer değiştirir.

1	3	2	8	9	2	5	6
1	3	2	8	2	9	5	6
1	3	2	8	2	5	9	6
1	3	2	8	2	5	6	9

Ters Elemanlar (Inversions)

Kabarcık sıralaması her zaman *ardışık* elemanların yerini değiştirir. Bu yüzden bu tip bir algoritmanın zaman karışıklığı *her zaman* en az $O(n^2)$ idir çünkü en kötü durumda diziyi sıralamak için toplam $O(n^2)$ değiştirme gereklidir.

Sıralama algoritmalarını incelerken işe yarar konseptlerden biri de **ters elemanlardır (inversion)**: Bir dizide öyle bir çift olsun ki elemanları $(array[a], array[b])$ olan bu çiftin elemanlarının sırası $a < b$ iken değerleri $array[a] > array[b]$ olmasına ters elemanlar deriz.

Örneğin aşağıdaki dizide:

1	2	2	6	3	5	9	8
---	---	---	---	---	---	---	---

üç tane ters eleman durumu vardır: $(6,3)$, $(6,5)$ ve $(9,8)$. Dizide bulunan ters eleman sayısı diziyi sıralamak için yapılması gereken işi gösterir. Dizide herhangi bir ters eleman durumu kalmayınca dizi sıralanmış olur. Ama, n elemanlı dizide elemanlar istenenin tam tersi sıradaysa ters eleman durumu miktarı maksimum olur:

$$1 + 2 + \dots + (n - 1) = \frac{n(n - 1)}{2} = O(n^2)$$

Yanlış sırada bulunan ardışık iki elemanın yerine değiştirmek tam olarak bir tane ters eleman durumunu kaldırır. Bu yüzden sadece ardışık elemanları değiştiren algoritmalarda her adımda maksimum bir ters eleman durumu kaldırıldığından dolayı bu tip algoritmaların zaman karmaşıklığı en az $O(n^2)$ olur.

$O(n \log n)$ Algoritmalar

Bir diziyi sadece ardışık eleman değiştirmeye bağlı kalmayarak $O(n \log n)$ 'lık algoritmalarla sıralayabiliriz. Bu tip algoritmalardan biri özyinelemeye (recursion) dayalı **birleştirme sıralamasıdır (merge sort)**.¹

Birleştirme sıralaması $\text{array}[a \dots b]$ alt dizisini aşağıdaki gibi sıralar:

1. Eğer $a = b$ ise, hiçbir şey yapma çünkü alt dizisi zaten sıralıdır.
2. Orta elemanın pozisyonunu hesapla: $k = \lfloor (a + b)/2 \rfloor$.
3. Özyinelemeli bir şekilde $\text{array}[a \dots k]$ alt dizisini sırala.
4. Özyinelemeli bir şekilde $\text{array}[k + 1 \dots b]$ alt dizisini sırala.
5. Sıralı alt dizileri yani $\text{array}[a \dots k]$ ve $\text{array}[k + 1 \dots b]$ dizilerini sıralı bir alt dizi şeklinde yandaki gibi $\text{array}[a \dots b]$ *sırala*.

Birleştirme sıralaması her seferinde alt dizinin eleman sayısını yarıya indirdiği için verimli bir algoritmadır. Algoritmadaki özyineleme $O(\log n)$ seviyeden oluşup her seviye işlemek $O(n)$ zaman alır. $\text{array}[a \dots k]$ $\text{array}[k + 1 \dots b]$ alt dizilerini zaten sıralı oldukları için linear zamanda birleştirmek mümkündür.

Örneğin aşağıdaki diziyi sıralarken:

1	3	6	2	8	2	5	9
---	---	---	---	---	---	---	---

Dizi aşağıdaki gibi iki alt diziye bölünecektir:

1	3	6	2
8	2	5	9

Sonrasında bu alt diziler özyinelemeli bir şekilde aşağıdaki gibi sıralanacaktır:

¹[47] kaynağına göre birleştirme sıralaması J. von Neumann tarafından 1945 tarihinde icat edilmiştir.

1	2	3	6
---	---	---	---

2	5	8	9
---	---	---	---

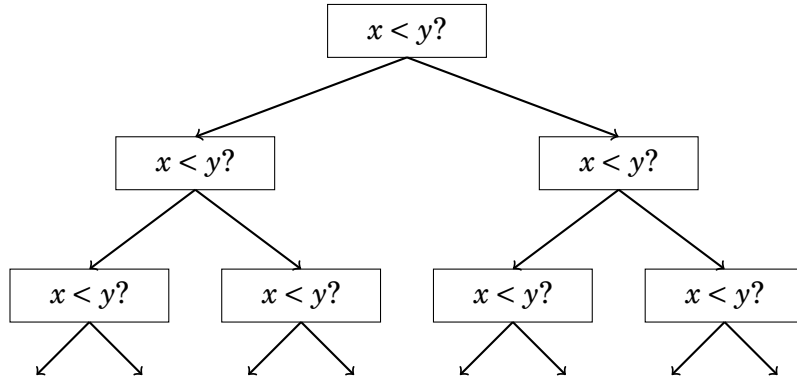
En sonda, algoritma sıralanmış alt dizileri birleştirip sıralanmış halde dizinin son halini verir.

1	2	2	3	5	6	8	9
---	---	---	---	---	---	---	---

Alt Sınırı (Lower Bound) Sıralamak

Bir diziyi $O(n \log n)$ süreden daha hızlı şekilde sıralamak mümkün müdür? Eğer kendimizi dizi elemanlarını birbirleriyle karşılaştıran sıralama algoritmalarıyla sınırlarsak bunun mümkün *olamayacağını* görürüz.

Zaman karışıklığının alt sınırı sıralamayı iki elemanın karşılaştırmasının dizi hakkında daha fazla bilgi verebileceği bir işlem olarak düşünülerek kanıtlanabilir. Bu işlem aşağıdaki şekilde bir ağaç oluşturur:



Burada " $x < y$?" ifadesi, bazı x ve y elemanlarının karşılaştırıldığını gösterilir. Eğer $x < y$ ise, işlem soldan devam ederken ifade doğru değilse işlem sağdan devam eder.

İşlemin sonuçları kullanılarak diziyi $n!$ yolla sıralamak mümkündür. Bu yüzden bu ağacın yüksekliği en az

$$\log_2(n!) = \log_2(1) + \log_2(2) + \dots + \log_2(n).$$

olmalıdır. Böylece bu toplamın alt sınırını son $n/2$ elemanı alıp her elemanın değerini $\log_2(n/2)$ değerine değiştirerek bulabiliriz. Bu ortalama olarak:

$$\log_2(n!) \geq (n/2) \cdot \log_2(n/2),$$

değerini verir ve böylece ağacın yüksekliği ile sıralama algoritmasındaki minimum adım sayısı en kötü durumda $n \log n$ olur.

Sayarak Sıralama (Counting Sort)

$n \log n$ alt tabanı dizi elemanlarını karşılaştırmak yerine başka bilgiler kullanan algoritmaları kapsamaz. Bu tip algoritmalarından biri **sayarak sıralama (counting sort)** olup $O(n)$ zamanda sıralar. Bu algoritma dizideki her elemanın değerinin $0 \dots c$ arasında olup $c = O(n)$ ifadesi kabul edilerek kullanılabilir.

Algoritma bir *değer tutma* dizisi oluşturur. Bu dizinin indisleri orijinal dizilerin elemanıdır. Algoritma, orijinal diziden geçerken her elemanın dizide kaç defa bulunduğunu tutar.

Örneğin aşağıdaki dizinin:

1	3	6	9	9	3	5	9
---	---	---	---	---	---	---	---

değer tutma dizisi aşağıdaki gibi olur:

1	2	3	4	5	6	7	8	9
1	0	2	0	1	1	0	0	3

Örneğin değer tutma dizisinde bulunan 3. pozisyonundaki eleman 2 değerine sahiptir çünkü 3 elemanı orijinal dizide 2 defa geçmektedir.

Değer tutma dizisinin oluşturulması $O(n)$ zaman alır. Bundan sonra sıralı dizi $O(n)$ zamanda oluşturulabilir çünkü her elemanın dizide kaç defa geçtiği değer tutma dizisinden alınabilir. Böylece sayarak sıralatma algoritmasının zaman karmaşıklığı $O(n)$ olur.

Sayarak sıralama çok verimli bir algoritma olmasına rağmen sadece sabit c ifadesi yeterince küçük olduğu zaman kullanılabilir ki böylece dizinin elemanları değer tutma dizisinde indis olarak kullanılabilir.

3.2 C++ Dilinde Sıralama

Yarışma sırasında kendinizin sıralama algoritması kodlaması gereksizdir çünkü zaten kodlama dillerinde verimli sıralama algoritmaları hazır halde bulunmaktadır. Örneğin C++ standart kütüphanesinde bulunan `sort` algoritması dizileri veya başka data yapılarını sıralarken kullanılabilir.

Hazır fonksiyon kullanmanın çeşitli avantajları vardır. Öncelikle fonksiyonu yeniden yazmaya gerek olmadığı için zaman kazandırır. İkincisi de doğru ve verimlidir. Bu yüzden yüksek ihtimal kodlayacağınız sıralama algoritmasının hazır olanından daha iyi olacağı pek olası değil.

Bu bölümde C++ dilinin `sort` fonksiyonunu nasıl kullanabileceğimizi göreceğiz. Aşağıdaki kod vektörü artan sırada sıralar:

```
vector<int> v = {4,2,5,3,5,8,3};  
sort(v.begin(),v.end());
```

Sıralamadan sonra vektörün içeriği aşağıdaki gibi olacaktır: `[2,3,3,4,5,5,8]`. Normal sıralama şekli artan sıradadır ama ters sıralama aşağıdaki gibi yapılabilir:

```
sort(v.rbegin(),v.rend());
```

Sıradan bir dizi aşağıdaki gibi sıralanabilir:

```
int n = 7; // array size  
int a[] = {4,2,5,3,5,8,3};  
sort(a,a+n);
```

Aşağıdaki kod s yazısını(string) sıralar:

```
string s = "monkey";  
sort(s.begin(), s.end());
```

Bir yazıyı sıralamak içindeki karakterleri alfabetik olarak sıralamak anlamına gelir. Örneğin "monkey" yazısı "ekmnoy" haline gelir.

Karşılaştırma İşlemleri (Comparison Operators)

sort fonksiyonu için istenilen data tipindeki elemanların sıralanması için bir **karşılaştırma işlemine** ihtiyaç duyar. Sıralama yapılırken bu işlem, iki elemanın sırası bulunması gerektiği zaman kullanılacaktır.

Çoğu C++ data tipinin(int vb.) hazır halde karşılaştırma işlemi bulunmaktadır ve bu elemanlar otomatik şekilde sıralanabilir. Örneğin sayılar, değerlerine göre sıralanırken yazılar alfabetik sıralarına göre sıralanırlar.

Çiftler (pair) öncelikle ilk elemanlarına göre sıralanırlar. (first). Eğer iki çiftin ilk elemanları eşitse ikinci elemanlarına göre sıralanırlar. (second):

```
vector<pair<int,int>> v;  
v.push_back({1,5});  
v.push_back({2,3});  
v.push_back({1,2});  
sort(v.begin(), v.end());
```

Bundan sonra çiftlerin sırası: (1,2), (1,5) ve (2,3) olur.

Demetler (tuple), benzer şekilde öncelikle ilk elemanlarına göre sıralanırlar. Sonrasında ikinci elemanlarına şeklinde sıralanır. ²

```
vector<tuple<int,int,int>> v;  
v.push_back({2,1,4});  
v.push_back({1,5,3});  
v.push_back({2,1,3});  
sort(v.begin(), v.end());
```

Bundan sonra demetlerin sırası (1,5,3), (2,1,3) ve (2,1,4) şeklinde olmaktadır.

Kullanıcı Tanımlı Yapılar (User-defined structs)

Kullanıcıların tanımladığı yapıların otomatik olarak bir karşılaştırma işlemi yoktur. Bu yüzden bu işlem yapının (struct) içinde operator< diye bir fonksiyon adında tanımlanmalıdır. Bu işlemin parametresi aynı tipten başka bir element olmalıdır. Bu işlem eğer şu anki eleman parametrede bulunan elementten küçükse true vermesi gerekirken büyük olduğunda da false vermesi gerekir.

²Bazı eski derleyicilerde demet tanımlanırken süslü parantez yerine make_tuple fonksiyonu kullanılmalıdır. (Örneğin {2,1,4} yerine make_tuple(2,1,4) ifadesiyle demet tanımlanmalıdır.)

Örneğin `P`, bir noktanın `x` ve `y` koordinatlarını içeren bir yapıdır. Karşılaştırma işlemi öyle bir tanımlanmalıdır ki noktalar öncelikle `x` koordinatlarına göre sıralanması gerekip ikincil olarak da `y` koordinatlarına göre sıralanması gerekir.

```
struct P {
    int x, y;
    bool operator<(const P &p) {
        if (x != p.x) return x < p.x;
        else return y < p.y;
    }
};
```

Karşılaştırma Fonksiyonları (Comparison Functions)

`sort` fonksiyonunu geri çağırım (callback) şeklinde dıştan bir **karşılaştırma fonksiyonu** ile çalıştırmak mümkündür. Örneğin aşağıdaki `comp` fonksiyonu yazıları öncelikle uzunluklarına göre karşılaştırır ve eş uzunlukta onları alfabetik sıralarına göre sıralar.

```
bool comp(string a, string b) {
    if (a.size() != b.size()) return a.size() < b.size();
    return a < b;
}
```

Şimdi bir yazı vektörü aşağıdaki gibi sıralanabilir:

```
sort(v.begin(), v.end(), comp);
```

3.3 İkili Arama Algoritması (Binary Search)

Genelde dizinin içindeki bir elemanı aramak için bütün dizideki elemanları dolaşan bir `for` döngüsü oluşturulur. Örneğin aşağıdaki kod dizide `x` elemanını arar:

```
for (int i = 0; i < n; i++) {
    if (array[i] == x) {
        // x i. indiste bulundu
    }
}
```

Bu çözümün zaman karmaşıklığı en kötü durumda dizideki bütün elemanları kontrol etmek gerekeceği için $O(n)$ idir. Eğer elemanların sırası karışıkysa bu aynı zamanda en iyi çözümdür çünkü dizide `x` elemanını bulabileceğimiz yer hakkında ekstradan bir bilgi verilmemiştir.

Fakat, dizi *sıralıysa* (*sorted*) bu durum değişir. Bu durumda aramayı çok daha hızlı yapabiliriz çünkü elemanların sırası elemanı ararken bize yardımcı

olur. Aşağıdaki **ikili arama algoritması** sayesinde sıralı dizide istenen elemanı $O(\log n)$ sürede bulabilmek mümkündür.

1. Yöntem

İkili Arama algoritmasını koda dökmenin klasik kolayı sözlükte kelime aramaya benzer. Arama dizide aktif bir bölge tutar. Bu aktif bölge başta bütün elemanları kapsarken her adımla beraber bu aktif bölgenin yarısına düşer.

Her adımda arama aktif bölgenin orta elemanını kontrol eder. Eğer ortadaki eleman istenen elemansa arama biter. Ama değilse ortadaki elemanın değerine göre arama özyinelemeli bir şekilde orta elemanın ya sağ tarafında ya da sol tarafında devam eder.

Yukarıdaki fikir aşağıdaki gibi koda yazılabilir:

```
int a = 0, b = n-1;
while (a <= b) {
    int k = (a+b)/2;
    if (array[k] == x) {
        //x, k. indiste bulundu
    }
    if (array[k] > x) b = k-1;
    else a = k+1;
}
```

Bu kodda aktif bölge $a \dots b$ olup ilk başta $0 \dots n-1$ bölgesidir. Algoritma her adımda bölgenin büyüklüğünü yarıya indirir ve böylece zaman karmaşıklığı $O(\log n)$ olur.

2. Yöntem

İkili Arama algoritmasını yazma yollarından bir başkası ise dizideki elemanları verimli bir şekilde geçmekten geçer. Buradaki fikir istenen elemana yaklaşırken büyüklüğü azalan zıplamalar yapmaktır.

Arama dizinin solundan sağına doğru gider ve ilk baştaki zıplamanın uzunluğu $n/2$ olur. Her adımda zıplamanın uzunluğu yarısına iner. Örneğin $n/2$, $n/4$, $n/8$, $n/16$ şeklinde gider. Bu durum zıplama uzunluğu 1 olana kadar devam eder. Zıplamalardan sonra ya istenen elemanı dizide buluruz ya da bu elemanın dizide bulunmadığını anlarız.

Yukarıdaki fikir aşağıdaki gibi koda yazılabilir:

```
int k = 0;
for (int b = n/2; b >= 1; b /= 2) {
    while (k+b < n && array[k+b] <= x) k += b;
}
if (array[k] == x) {
    // x, k. indiste bulundu
}
```

Arama sırasında b değişkeni şu anki zıplama uzunluğunu tutar. Bu algoritmanın zaman karmaşıklığı $O(\log n)$ idir çünkü koddaki `while` döngüsü her zıplama uzunluğu için en fazla iki defa yapılabilir.

C++ Fonksiyonları

C++ standart kütüphanesi aşağıdaki ikili arama algoritmasına dayalı olup logaritmik zamanda çalışan fonksiyonları içerir:

- `lower_bound` değeri en az x olan ilk dizi elemanını gösteren bir işaretçi(pointer) döner.
- `upper_bound` değeri x 'in değerinden büyük olan ilk elemanı gösteren bir işaretçi döner.
- `equal_range` yukarıdaki iki işaretçiye döner.

Bu fonksiyonlar dizinin sıralı olduğunu varsayar. Eğer istenilen eleman dizide yoksa dönülen işaretçi son dizi elemanından bir sonraki elemanı döner. Örneğin aşağıdaki kod dizide x değerine sahip bir eleman içerip içermediğini kontrol eder:

```
auto k = lower_bound(array,array+n,x)-array;
if (k < n && array[k] == x) {
    // x, k. indiste bulundu
}
```

Aşağıdaki kod ise değeri x olan eleman sayısını döner:

```
auto a = lower_bound(array, array+n, x);
auto b = upper_bound(array, array+n, x);
cout << b-a << "\n";
```

`equal_range` ifadesi kullanılarak kod daha kısa halde yazılabilir:

```
auto r = equal_range(array, array+n, x);
cout << r.second-r.first << "\n";
```

En Küçük Çözümü Bulmak

İkili Arama algoritmasının önemli kullanımlarından biri *fonksiyonun* değiştiği pozisyonu bulmaktır. Varsayalım ki bir problem için uygun bir çözüm olan en küçük k değerini bulmak istiyoruz. Bize $ok(x)$ diye bir fonksiyon verilmiştir ve bu fonksiyon eğer x geçerli bir çözümse `true` yani doğru döndürecekken geçerli bir çözüm değilse de `false` yani yanlış döndürecektir. Ekstradan $ok(x)$ fonksiyonunun $x < k$ olduğu zaman `false` yani yanlış döndüreceğini ve $x \geq k$ olduğu zaman da `true` yani doğru döndüreceğini biliyoruz. Bu durum aşağıdaki gibi olur:

x	0	1	...	$k-1$	k	$k+1$...
$ok(x)$	false	false	...	false	true	true	...

Şimdi k değeri ikili arama algoritması ile bulunabilir:

```
int x = -1;
for (int b = z; b >= 1; b /= 2) {
    while (!ok(x+b)) x += b;
}
int k = x+1;
```

Arama $ok(x)$ fonksiyonunun `false` döndürecek en büyük x değerini bulur. Böylece sonraki değer $k = x + 1$ olup $ok(x)$ fonksiyonunu `true` döndürecek en küçük değeri bulur. İlk zıplama uzunluğunun yani z değerinin yeterince büyük olması lazım. Örneğin z değeri olarak $ok(z)$ fonksiyonunu `true` döndüren bir değer vermemiz gerekir.

Algoritma ok fonksiyonunu $O(\log z)$ defa çağırır ve böylece toplam zaman karmaşıklığı ok fonksiyonuna bağlı olur. Örneğin fonksiyon $O(n)$ zamanda çalışıyorsa toplam zaman karmaşıklığı $O(n \log z)$ olur.

En Büyük Değeri Bulmak

İkili Arama algoritması başta artan ve sonra azalan bir fonksiyondaki en büyük değeri bulmak için kullanılabilir. Bizim amacımız öyle bir k pozisyonu bulmak ki

- $x < k$ olduğu zaman $f(x) < f(x + 1)$ ve
- $x \geq k$ olduğu zaman $f(x) > f(x + 1)$ olmalıdır.

Buradaki fikir ikili arama algoritmasını kullanarak $f(x) < f(x + 1)$ ifadesini sağlayan en büyük x değerini bulmaktır. Bu $k = x + 1$ olacağını söyler çünkü $f(x + 1) > f(x + 2)$ olur. Aşağıdaki kod istediğimiz işlemi yapar:

```
int x = -1;
for (int b = z; b >= 1; b /= 2) {
    while (f(x+b) < f(x+b+1)) x += b;
}
int k = x+1;
```

Burada dikkat edilmesi gereken, sıradan bir ikili arama algoritmasında olan aksine burada bir fonksiyonun ardışık ifadelerin aynı değere sahip olmaması gerekir. Öyle bir durumda aramanın nasıl devam edileceği bilinemez.

Bölüm 4

Veri Yapıları

Veri yapısı, bilgisayarın hafızasında veri saklamak için bir yoldur. Ele alınan problem için uygun bir veri yapısı seçimi yapmak önemlidir çünkü her veri yapısının kendine göre avantajları ve dezavantajları bulunmaktadır. Bu noktada cevaplanması gereken ana soru, hangi işlemlerin seçtiğimiz veri yapısında verimli olacaktır.

Bu bölümde C++ standart kütüphanesindeki en önemli veri yapılarını tanıttacaktır. Standart kütüphane zamandan tasarruf sağlayacağı için mümkün oldukça onu kullanmaya özen gösterilmelidir. Kitabın ilerleyen bölümlerinde standart kütüphanede bulunmayan daha sofistike veri yapılarını da öğreneceğiz.

4.1 Dinamik Diziler

Dinamik dizi, programın çalışması sırasında boyutu değiştirilebilen dizilere verilen addır. C++'daki en popüler dinamik dizi, **vector** yapısıdır. Vektör yapısı, neredeyse sıradan bir diziymiş gibi kullanılabilir.

Aşağıdaki kod, boş bir vektör oluşturur ve bu vektöre üç eleman ekler:

```
vector<int> v;  
v.push_back(3); // [3]  
v.push_back(2); // [3,2]  
v.push_back(5); // [3,2,5]
```

Bunun ardından, elemanlara sanki sıradan bir dizidelermiş gibi erişilebilir:

```
cout << v[0] << "\n"; // 3  
cout << v[1] << "\n"; // 2  
cout << v[2] << "\n"; // 5
```

size fonksiyonu vektörde bulunan eleman sayısını döndürür. Aşağıdaki kod, vektör boyunca ilerleyerek içerdiği bütün elemanları yazdırır:

```
for (int i = 0; i < v.size(); i++) {  
    cout << v[i] << "\n";  
}
```

Bir vektör boyunca ilerlemenin daha kısa bir yolu aşağıdaki gibidir:

```
for (auto x : v) {  
    cout << x << "\n";  
}
```

back fonksiyonu vektörde bulunan son elemanı döndürürken pop_back fonksiyonu son elemanı vektörden çıkarır:

```
vector<int> v;  
v.push_back(5);  
v.push_back(2);  
cout << v.back() << "\n"; // 2  
v.pop_back();  
cout << v.back() << "\n"; // 5
```

Aşağıdaki kod, beş elemanlı bir vektör oluşturur:

```
vector<int> v = {2,4,2,5,1};
```

Vektör oluşturma'nın bir başka yolu da toplam eleman sayısını ve her elemanın ilk başta alacağı bir değeri vermektir:

```
// size 10, initial value 0  
vector<int> v(10);
```

```
// size 10, initial value 5  
vector<int> v(10, 5);
```

Vektör yapısının içeride kullandığı implementasyon aslında sıradan bir dizidir. Eğer vektörün boyutu büyür ve bu dizi küçük kalmaya başlarsa, hafızada yeni bir dizi ayrılır ve bütün elemanlar bu yeni diziye taşınır. Bu çok sık gerçekleşmediği için¹ push_back operasyonunun ortalama zaman karışıklığı $O(1)$ 'dir.

string veri yapısı da neredeyse bir vektör gibi kullanılabilen bir dinamik dizidir. Ayrıca, diğer veri yapılarında bulunmayan bazı söz dizim (syntax) kurallarına sahiptir. Stringler + sembolü kullanılarak birleştirilebilirler. substr(k, x) fonksiyonu, k pozisyonunda başlayan x uzunluğundaki alt stringi (substring) döndürür. find(t) fonksiyonu ise t alt stringiyle ilk karşılaşılacak pozisyonu döndürür.

Aşağıdaki kod bazı string işlemlerini göstermektedir:

```
string a = "hatti";  
string b = a+a;  
cout << b << "\n"; // hattihatti  
b[5] = 'v';  
cout << b << "\n"; // hattivatti
```

¹(Ç. N.) İnternet sistemleri dizayn edilirken kullanılan standart kütüphanelerdeki dinamik dizilerin boyutunun hangi değerde artacağı kamuya açık bir bilgi olduğundan bazı hacker'ların bu duruma hoş olmayan bir istisna oluşturduğunu söyleyebiliriz.

```
string c = b.substr(3,4);  
cout << c << "\n"; // tiva
```

4.2 Küme (Set) Yapıları

Küme, elemanlar topluluğu barındıran bir veri yapısıdır. Kümelere ait basit işlemler eleman ekleme (insertion), arama (search) ve kaldırmadır (removal).

C++ standart kütüphanesi iki ayrı küme implementasyonu içerir: **set** yapısı dengeli bir ikili ağaç (balanced binary tree) üstüne kuruludur ve işlemleri $O(\log n)$ sürede çalışır. **unordered_set** yapısı ise kıyım (hashing) kullanır ve işlemleri ortalama olarak $O(1)$ sürede çalışır.

Hangi küme implementasyonunun kullanılacağı çoğu zaman bir kişisel tercih meselesidir. **set** yapısının faydası, elemanların sırasını muhafaza etmesi ve **unordered_set** yapısında bulunmayan fonksiyonlara sahip olmasıdır. **unordered_set** yapısının faydası ise ötekine kıyasla daha verimli çalışabileceğidir.

Aşağıdaki kod, tam sayılar içeren bir küme oluşturur ve bazı işlemleri gösterir. **insert** fonksiyonu kümeye bir eleman ekler, **count** fonksiyonu bir elemanın kümede toplam kaç defa bulunduğunu döndürür, **erase** fonksiyonu ise kümeden bir eleman çıkarır.

```
set<int> s;  
s.insert(3);  
s.insert(2);  
s.insert(5);  
cout << s.count(3) << "\n"; // 1  
cout << s.count(4) << "\n"; // 0  
s.erase(3);  
s.insert(4);  
cout << s.count(3) << "\n"; // 0  
cout << s.count(4) << "\n"; // 1
```

Kümeler genel itibarıyla sanki bir vektörmüş gibi kullanılabilir ancak elemanlarına [] notasyonu ile erişmek mümkün değildir. Aşağıdaki kod bir küme yaratır, bulundurduğu eleman sayısını yazdırır ve küme üzerinde ilerleyerek her elemanını yazdırır:

```
set<int> s = {2,5,6,8};  
cout << s.size() << "\n"; // 4  
for (auto x : s) {  
    cout << x << "\n";  
}
```

Kümelerin önemli bir özelliği, bütün elemanlarının *farklı* olmasıdır. Yani, **count** fonksiyonu her zaman ya 0 döndürür (eleman kümede bulunmamaktadır), ya da 1 döndürür (eleman kümede bulunmaktadır). **insert** fonksiyonu da eğer bir eleman halihazırda kümede bulunmaktaysa o elemanı asla kümeye eklemez.

Aşağıdaki kod bunu göstermektedir:

```
set<int> s;
s.insert(5);
s.insert(5);
s.insert(5);
cout << s.count(5) << "\n"; // 1
```

C++ aynı zamanda `multiset` ve `unordered_multiset` veri yapılarını da içerir. Bu veri yapıları `set` ve `unordered_set` gibi çalışır ancak bir eleman birden fazla sayıda bulunabilir. Örneğin aşağıdaki kodda, 5 sayısının her üç kopyası da çoklu kümeye eklenmiştir:

```
multiset<int> s;
s.insert(5);
s.insert(5);
s.insert(5);
cout << s.count(5) << "\n"; // 3
```

`erase` fonksiyonu verilen elemanın bütün kopyalarını çoklu kümeden siler:

```
s.erase(5);
cout << s.count(5) << "\n"; // 0
```

Çoğu zaman yalnızca bir kopyanın kaldırılmasını isteriz, bu da aşağıda gösterildiği gibi yapılabilir:

```
s.erase(s.find(5));
cout << s.count(5) << "\n"; // 2
```

4.3 Map Yapıları

Bir **map**, anahtar-değer çiftlerinden (key-value-pairs) oluşan genelleştirilmiş bir dizidir. Sıradan bir dizideki anahtarlar (n dizinin büyüklüğü olduğu kabul edilirse) her zaman ardışık $0, 1, \dots, n-1$ sayıları olurken map'teki anahtarlar herhangi bir tip veri tipi olabilir ve ardışık değer olmak zorunda değildir.

C++ standart kütüphanesi küme implementasyonuna karşılık gelen iki harita implementasyonu içerir: `map` yapısı dengeli iki ağaç yapısından oluşup elemanlara ulaşması $O(\log n)$ zaman alırken `unordered_map` yapısı hashleme kullanır ve elemanlara ulaşması *ortalamada* $O(1)$ zaman alır.

Aşağıdaki kod anahtarların yazı olduğu ve değerlerin tamsayı olduğu bir map yapısı oluşturur:

```
map<string,int> m;
m["monkey"] = 4;
m["banana"] = 3;
m["harpsichord"] = 9;
```



```
sort(v.begin(), v.end());  
reverse(v.begin(), v.end());  
random_shuffle(v.begin(), v.end());
```

Bu fonksiyonlar sıradan bir dizide de kullanılır. Bu durumda fonksiyona iteratorler yerine işaretçiler (pointer) verilir:


```
sort(a, a+n);
reverse(a, a+n);
random_shuffle(a, a+n);
```

Set İteratörleri

İteratörler genelde bir set'in elemanlarına erişmek için kullanılır. Aşağıdaki kod set'teki en küçük elemanı gösteren bir `it` iteratörü oluşturur:

```
set<int>::iterator it = s.begin();
```

Kodu yazmanın daha kısa yolu şöyledir:

```
auto it = s.begin();
```

İteratörün gösterdiği elemana `*` sembolü ile ulaşabiliriz. Örneğin aşağıdaki kod set'in ilk elemanını yazdırır:

```
auto it = s.begin();
cout << *it << "\n";
```

İteratörler `++` (ileri) ve `--` (geri) operatörleri ile hareket ettirilebilir. Bunlardan ilki iteratörü bir adım ileri, ikincisi de bir adım geri götürür.

Aşağıdaki kod bütün elemanları artan sırada sıralar:

```
for (auto it = s.begin(); it != s.end(); it++) {
    cout << *it << "\n";
}
```

Aşağıdaki kod set'teki en büyük elemanı yazdırır:

```
auto it = s.end(); it--;
cout << *it << "\n";
```

`find(x)` fonksiyonu değeri x olan bir elemanı gösteren bir iteratör döndürür. Fakat eğer set x içermiyorsa iteratör `end`'i gösterir.

```
auto it = s.find(x);
if (it == s.end()) {
    // x bulunamad
}
```

`lower_bound(x)` fonksiyonu, setteki değeri *en az* x olan en küçük elemanı döndürür ve `upper_bound(x)` fonksiyonu da setteki değeri x 'ten *büyük* olan en küçük elemanı döndürür. İki fonksiyonda da, eğer böyle bir eleman yoksa değer olarak `end` döndürülür. Bu fonksiyonlar, elemanların sırasını tutmayan `unordered_set` yapısında kullanılamaz.

Örneğin aşağıdaki kod, x 'e en yakın elemanı bulur:

```
auto it = s.lower_bound(x);
if (it == s.begin()) {
    cout << *it << "\n";
} else if (it == s.end()) {
    it--;
    cout << *it << "\n";
} else {
    int a = *it; it--;
    int b = *it;
    if (x-b < a-x) cout << b << "\n";
    else cout << a << "\n";
}
```

Kod set'in boş olmadığını varsayar ve `it` iteratörü ile bütün olası durumlardan geçer. İlk, iteratör değeri en az x olan en küçük elemanı gösterir. Eğer `it` `begin`'e eşitse şu anki eleman x 'e en yakındır. Eğer `it` `end`'e eşitse set'teki en büyük eleman x 'e en yakındır. Eğer önceki durumların hiçbiri yoksa, x 'e en yakın eleman ya `it` iteratörünün gösterdiği elemandır ya da ondan bir önceki elemandır.

4.5 Diğer Yapılar

Bitset

Bir **bitset** her değeri ya 0 ya da 1 olan bir dizidir. Örneğin aşağıdaki kod 10 elemandan oluşan bir **bitset** oluşturur:

```
bitset<10> s;
s[1] = 1;
s[3] = 1;
s[4] = 1;
s[7] = 1;
cout << s[4] << "\n"; // 1
cout << s[5] << "\n"; // 0
```

Bitset kullanmanın yararı sıradan dizilere göre daha hafıza yiyor olmasıdır çünkü bitsetteki her eleman sadece bir bit hafıza yer yemektedir. Örneğin eğer n tane bit `int` dizisinde tutulduysa $32n$ bitlik hafıza kullanırken buna karşılık gelen bir **bitset** sadece n bitlik hafıza kullanır. Bununla beraber bir bitsetteki değerler verimli bir şekilde bit operatörleri ile manipüle edilebilir ki böylece algoritmaları bitsetlerle optimize edebiliriz.

Aşağıdaki kod yukarıdaki bitseti oluşturma'nın başka bir yolunu gösteriyor:

```
bitset<10> s(string("0010011010")); // 00sadan sola
cout << s[4] << "\n"; // 1
cout << s[5] << "\n"; // 0
```

count fonksiyonu bitsetteki 1 miktarını döndürmektedir:

```
bitset<10> s(string("0010011010"));
cout << s.count() << "\n"; // 4
```

Aşağıdaki kod bit operasyonlarını kullanmanın örneklerini gösterir:

```
bitset<10> a(string("0010110110"));
bitset<10> b(string("1011011000"));
cout << (a&b) << "\n"; // 0010010000
cout << (a|b) << "\n"; // 1011111110
cout << (a^b) << "\n"; // 1001101110
```

Deque

Deque dinamik bir dizi olup büyüklüğü dizinin her iki tarafından da değiştirilebilir. Vektördeki gibi, deque'de de `push_back` ve `pop_back` fonksiyonlarını kullanabiliriz fakat aynı zamanda vektörde bulunmayan `push_front` ve `pop_front` fonksiyonlarını kullanabiliriz.

Bir deque aşağıdaki gibi kullanılabilir:

```
deque<int> d;
d.push_back(5); // [5]
d.push_back(2); // [5,2]
d.push_front(3); // [3,5,2]
d.pop_back(); // [3,5]
d.pop_front(); // [5]
```

Deque'in iç implementasyonu vektörden daha karışık olduğu için deque, vektörden daha yavaştır. Yine de hem eleman ekleme hem de eleman çıkarma iki uçta da ortalama $O(1)$ zaman alır.

Yığın (Stack)

Yığın iki operasyonu $O(1)$ zamanda yapan bir veri yapısıdır: en üste eleman eklemek ve en üstten eleman çıkarmak. Yığında sadece en üstteki elemana ulaşmak mümkündür.

Aşağıdaki kod bir yığının nasıl kullanabileceğini gösterir:

```
stack<int> s;
s.push(3);
s.push(2);
s.push(5);
cout << s.top(); // 5
s.pop();
cout << s.top(); // 2
```

Kuyruk (Queue)

Kuyruk iki operasyonu $O(1)$ zamanda yapan bir veri yapısıdır: kuyruğun sonuna eleman eklemek ve kuyruğun başındaki elemanı çıkarmak. Bir kuyruğun sadece ilk ve son elemanına ulaşmak mümkündür.

Aşağıdaki kod bir kuyruğun nasıl kullanılabileceğini gösterir:

```
queue<int> q;
q.push(3);
q.push(2);
q.push(5);
cout << q.front(); // 3
q.pop();
cout << q.front(); // 2
```

Priority queue

Priority queue bir elemanlar kümesini tutar. Desteklenen operasyonlar ekleme, kuyruğun tipine göre erişme ile ya minimum ya da maksimum elemanı çıkarmaktır. Ekleme ve çıkarma $O(\log n)$ zaman alırken erişme sadece $O(1)$ zaman alır.

Ordered set verimli bir şekilde priority queue'nin bütün operasyonlarını destekliyor olsa da priority queue kullanmanın yararı daha küçük sabit çarpanlara sahip olmasıdır. Bir priority queue, heap yapısıyla oluşturulur ki bu ordered set'te kullanılan dengeli ikili ağaca göre daha basittir.

Normalde, bir C++ priority queue'sındaki elemanlar büyükten küçüğe sıralanır ve böylece kuyruktaki en büyük elemanı bulup silebilmek mümkündür. Aşağıdaki kod bunu gösterir:

```
priority_queue<int> q;
q.push(3);
q.push(5);
q.push(7);
q.push(2);
cout << q.top() << "\n"; // 7
q.pop();
cout << q.top() << "\n"; // 5
q.pop();
q.push(6);
cout << q.top() << "\n"; // 6
q.pop();
```

Eğer en küçük elemanı bulup veya silebilen bir priority queue oluşturmak istiyorsak aşağıdaki gibi yapabiliriz:

```
priority_queue<int, vector<int>, greater<int>>> q;
```

Policy Tabanlı Veri Yapıları (Policy-Based Data Structures)

g++ derleyicisi C++ standart kütüphanesinde bulunmayan bazı data yapıları içerir. Bu tip yapılara *policy tabanlı* veri yapısı denir. Bu yapıları kullanmak için kodun başına şu satırlar eklenmelidir:

```
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
```

Bundan sonra, bir veri yapısı olan `indexed_set` yapısını kullanırız ki bu set gibidir fakat dizi gibi indislenebilir haldedir. `int` değerleri için tanımı aşağıdaki gibidir:

```
typedef tree<int,null_type,less<int>,rb_tree_tag,
            tree_order_statistics_node_update> indexed_set;
```

Şimdi bir set'i aşağıdaki gibi oluştururuz. Now we can create a set as follows:

```
indexed_set s;
s.insert(2);
s.insert(3);
s.insert(7);
s.insert(9);
```

Bu set'in özelliği elemanlar sanki sıralı bir dizideymiş gibi indislerine erişebiliyor olmamızdır. `find_by_order` fonksiyonu verilen pozisyonundaki bulunan elemanı gösterne bir iteratör döndürür:

```
auto x = s.find_by_order(2);
cout << *x << "\n"; // 7
```

`order_of_key` fonksiyonu da verilene elemanın pozisyonunu verir:

```
cout << s.order_of_key(7) << "\n"; // 2
```

Eğer eleman set'te yoksa elemanın normalde sette bulunacağı pozisyonu verir:

```
cout << s.order_of_key(6) << "\n"; // 2
cout << s.order_of_key(8) << "\n"; // 3
```

İki fonksiyon da logaritmik zamanda çalışır.

4.6 Comparison to sorting

Genelde bir problemi ya sıralama ile ya da data yapıları çözmek mümkündür. Bazen bu yaklaşımların asıl verimlilikleri arasında hatırı sayılır bir fark olabilir ki bu da zaman karmaşıklıklarında saklı olabilir.

Diyelim ki bize ikisi de n elemandan oluşan A ve B listeleri verildiğini düşünelim. Görevimiz her iki listeye de ait elemanların sayısını bulmak. Örneğin

listeler

$$A = [5, 2, 8, 9] \text{ and } B = [3, 2, 9, 5]$$

olursa cevap 3 olur çünkü iki listede de 2, 5 ve 9 sayıları bulunur.

Bunun düz bir çözümü bütün eleman çiftlerinden $O(n^2)$ zamanda gitmektir fakat şimdi daha verimli algoritmalarından konuşacağız:

1. Algoritma

A 'da bulunan elemanlardan oluşan bir set oluşturacağız ve sonrasında B 'deki elemanlardan dolaşp her eleman için A 'da da bulunup bulunmadığına bakacağız. Bu verimlidir çünkü A 'nın elemanları bir set'tedir. set yapısını kullanarak algoritmanın zaman karmaşıklığı $O(n \log n)$ olur.

2. Algoritma

Ordered set tutmak zorunlu değildir o yüzden set yapısı yerine unordered_set yapısı kullanabiliriz. Bu algoritmayı daha verimli yapmanın daha kolay bir yoludur çünkü sadece altta yapan veri yapısını değiştirmemiz gerekir. Bu yeni algoritmanın zaman karmaşıklığı $O(n)$ olur.

3. Algoritma

Veri yapıları yerine sıralayarak yapabiliriz. İlk başta hem A hem de B listelerini sıralarız. Sonrasında iki listeden de aynı zamanda geçeriz ve ortak elemanları buluruz. Sıralamanın zaman karmaşıklığı $O(n \log n)$ olur ve algoritmanın geriye kalan kısmı $O(n)$ zaman alır yani toplam zaman karmaşıklığı $O(n \log n)$ olur.

Verimlilik Karşılaştırması

Aşağıdaki tablo yukarıdaki algoritmaların n sayısının değiştiği zaman ve listedeki elemanların $1 \dots 10^9$ arasında rastgele sayılar oldukları zaman ne kadar verimli olduklarını gösterir:

n	1. Algoritma	2. Algoritma	3. Algoritma
10^6	1.5 s	0.3 s	0.2 s
$2 \cdot 10^6$	3.7 s	0.8 s	0.3 s
$3 \cdot 10^6$	5.7 s	1.3 s	0.5 s
$4 \cdot 10^6$	7.7 s	1.7 s	0.7 s
$5 \cdot 10^6$	10.0 s	2.3 s	0.9 s

1 ve 2. algoritmalar farklı set yapıları hariç aynıdır. Bu problemde bu seçim önemli bir etkiye sahiptir çünkü 2. algoritma 1. algoritmaya göre 4-5 kat daha hızlı çalışır.

Fakat sıralamayı kullanan 3. algoritma aralarındaki en verimli algoritmadır. Bu sadece 2. algoritmanın yarısı süresinde çalışmaktadır. İlginç bir şekilde hem 1 hem 3. algoritmanın zaman karmaşıklığı $O(n \log n)$ idir ama buna rağmen 3.

algoritma 10 kat daha hızlıdır. Bu durum sıralamanın basit bir işlem olmasından ve 3. algoritmanın başında sadece bir kere yapılmasından açıklanabilir. Diğer tarafta 1. algoritma, bütün algoritma sırasında karışık bir dengeli ikili ağaç yapısı kullanır.

Bölüm 5

Tam Arama (Complete Search)

Tam arama neredeyse her algoritma probleminde kullanılan bir metoddur. Buradaki fikir, kaba kuvvet (brute force) kullanarak bütün olası çözümleri oluşturup sonrasında probleme göre aralarında en iyi çözümü veya bütün çözümler sayılarak yapılır.

Tam arama bütün çözümler denenebiliyorsa işe yarar bir tekniktir çünkü bu aramayı koda dökmek kolaydır ve aynı zamanda her zaman doğru çözümü bulur. Eğer tam arama soru için çok yavaş kalıyorsa açgözlü (greedy) algoritmalar veya dinamik programlama gibi başka yöntemler gerekebilir.

5.1 Alt küme Oluşturmak (Generating Subsets)

n elemanlı bir dizinin bütün alt kümelerini oluşturmamız gerektiğini düşünelim. Örneğin $\{0,1,2\}$ kümesinin alt kümeleri \emptyset , $\{0\}$, $\{1\}$, $\{2\}$, $\{0,1\}$, $\{0,2\}$, $\{1,2\}$ and $\{0,1,2\}$ olur. Alt küme oluşturmak için iki yaygın yöntem vardır: Ya özyinelemeli (recursive) arama yaparız ya da tam sayıların bit gösterimlerini kullanırız.

1. Method

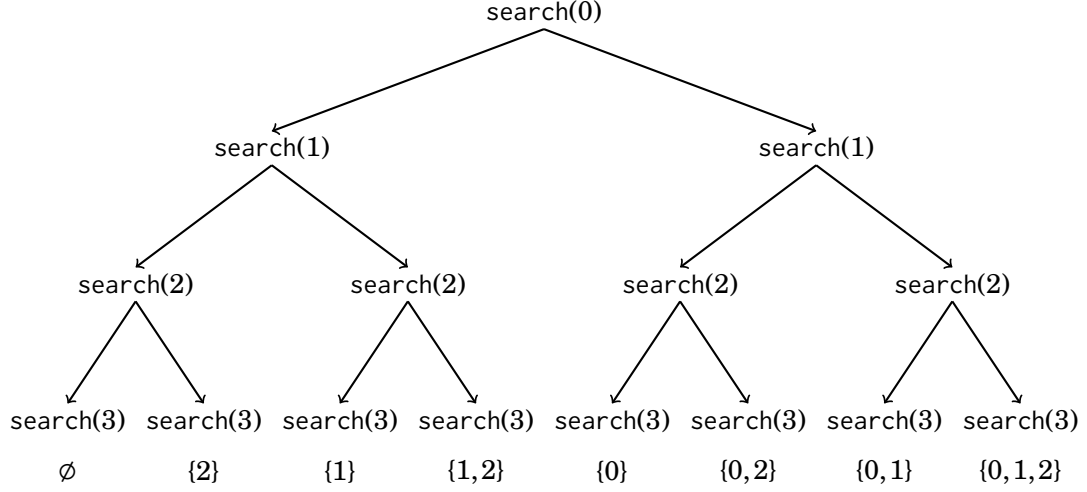
Bütün alt kümeleri oluşturmanın sık yollarından biri özyineleme yapmaktır. Aşağıdaki `search` fonksiyonu $\{0,1,\dots,n-1\}$ dizisinin alt kümelerini oluşturur.

Fonksiyon her alt kümenin elemanlarını tutan bir `subset` bulundurur. Arama, fonksiyon 0 parametresiyle çağrıldığı zaman başlar.

```
void search(int k) {
    if (k == n) {
        // alt kümeyi şile
    } else {
        search(k+1);
        subset.push_back(k);
        search(k+1);
        subset.pop_back();
    }
}
```

`search` fonksiyonu k parametresi ile çağrıldığı zaman k elemanını ekleyip eklemeyeceğine karar verir. Sonra kendisini $k + 1$ parametresiyle yeniden çağırır. Fakat $k = n$ olunca fonksiyon bütün elemanların dolaşıldığını ve bir alt küme oluşturulduğunu fark eder.

Aşağıdaki ağaç fonksiyon $n = 3$ olduğu zaman nasıl çalıştığını gösterir. Her zaman ya sol (k bu alt kümede bulunmaz.) veya sağ kolu (k bu alt kümede bulunur.) seçebiliriz.



2. Method

Alt küme oluşturmanın başka bir yolu işte tam sayıların (integer) bit gösterimlerinden yararlanmaktır. n elemanlık bir dizinin her alt kümesi n tane bitten oluşan bir dizi şeklinde gösterilebilir. Bunlar da $0 \dots 2^n - 1$ sayılarına denk gelmektedir. Bit dizisindeki "1"ler alt kümede hangi elemanların bulunduğunu gösterir.

Genelde son bit 0. elemana, sondan ikinci bit 1. elemana vb. denk gelir. Örneğin 25'in bit gösterimi 11001 olur ve bu {0,3,4} alt kümesine denk gelir.

Aşağıdaki kod n elemanlı bir dizinin alt kümelerinden geçer:

```

for (int b = 0; b < (1<<n); b++) {
    // process subset
}

```

Aşağıdaki kod verilen bit dizisine göre alt kümedeki elemanları nasıl bulabileceğimizi gösterir. Her alt kümeyi işlerken o alt kümede bulunan elemanlar için kod bir vektör oluşturur.

```

for (int b = 0; b < (1<<n); b++) {
    vector<int> subset;
    for (int i = 0; i < n; i++) {
        if (b & (1<<i)) subset.push_back(i);
    }
}

```

5.2 Permütasyon Oluşturmak (Generating Permutations)

n elemandan oluşan bir dizinin bütün permütasyonlarını bulmamız gerektiğini düşünelim. $\{0, 1, 2\}$ alt kümesinin permütasyonları $(0, 1, 2)$, $(0, 2, 1)$, $(1, 0, 2)$, $(1, 2, 0)$, $(2, 0, 1)$ ve $(2, 1, 0)$ olur. Yine iki yolla bulabiliriz: Ya özyinelemeli (recursive) ya da düz döngüyle (iteratively) şekilde buluruz.

1. Method

Alt kümeler gibi permütasyonlar da özyineleme kullanılarak oluşturulabilir. Aşağıdaki `search` fonksiyonu $\{0, 1, \dots, n-1\}$ setinin bütün permütasyonlarını bulur. Fonksiyon permütasyonu tutan bir `permutation` vektörü tutar ve arama fonksiyona parametre verilmeden başlanır.

```
void search() {
    if (permutation.size() == n) {
        // permütasyonu şile
    } else {
        for (int i = 0; i < n; i++) {
            if (chosen[i]) continue;
            chosen[i] = true;
            permutation.push_back(i);
            search();
            chosen[i] = false;
            permutation.pop_back();
        }
    }
}
```

Her fonksiyon çağrıldığında `permutation` vektörüne yeni bir eleman eklenir. `chosen` dizisi şu anda hangi elemanların permütasyonda bulunduğunu gösterir. Eğer `permutation` vektörünün büyüklüğü dizinin büyüklüğüne eşitse bir permütasyon oluşturulmuş anlamına gelir.

2. Method

Permütasyon oluşturma başka bir yolu ise $\{0, 1, \dots, n-1\}$ permütasyonu ile başlayıp artan bir şekilde sonraki permütasyonu oluşturan bir fonksiyon kullanmaktır. C++ standart kütüphanesinde olan `next_permutation` fonksiyonu bu amaçla kullanılabilir.

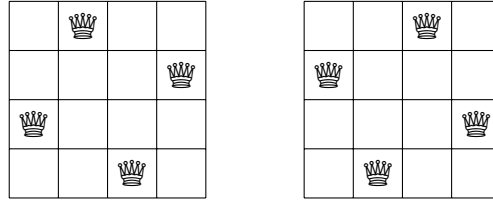
```
vector<int> permutation;
for (int i = 0; i < n; i++) {
    permutation.push_back(i);
}
do {
```

```
// permütasyonu uygula
} while (next_permutation(permutation.begin(), permutation.end()));
```

5.3 Geri İzleme (Backtracking)

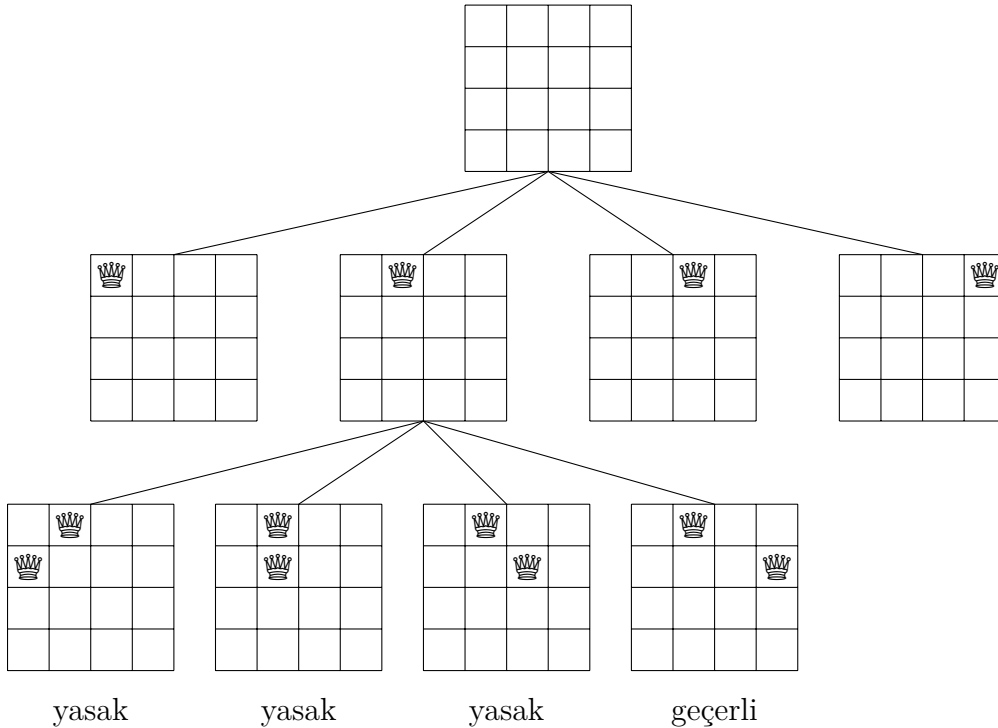
Geri izleme algoritması boş bir çözümle başlayıp çözümü adım adım büyütür. Arama özyinelemeli şekilde çözümü oluşturabilen bütün yolları dener.

Örneğin n veziri, $n \times n$ 'lik bir satranç tahtasında iki vezirin birbirini tehdit etmeyecek şekilde kaç farklı şekilde yerleştirebileceğimizi bulmamız gerekir. Örneğin $n = 4$ olduğu zaman iki çözüm vardır:



Problem geri izleme yöntemi ile bütün vezirleri tahtaya satır satır koyarak çözülebilir. Yani, her veziri kendinden önceki vezirleri tehdit etmeyecek şekilde ayrı ayrı satırlara koyulması gerekir. n vezir tahtaya koyulduğu zaman bir çözüm bulunmuş oluyor.

Örneğin $n = 4$ olduğu zaman geri izleme algoritması ile aşağıdaki yarım çözümler oluşur:



En alt seviyede ilk üç sıralama yasaktır çünkü burada vezirler birbirini tehdit etmektedir. Fakat 4. sıralama uygundur ve tahtaya iki tane daha vezir eklenerek tam çözüm bulunabilir. Tahtaya geri kalan 2 veziri bir yolla yerleştirebiliriz.

Bu algoritma şu şekilde koda dökülebilir:

```
void search(int y) {
    if (y == n) {
        count++;
        return;
    }
    for (int x = 0; x < n; x++) {
        if (column[x] || diag1[x+y] || diag2[x-y+n-1]) continue;
        column[x] = diag1[x+y] = diag2[x-y+n-1] = 1;
        search(y+1);
        column[x] = diag1[x+y] = diag2[x-y+n-1] = 0;
    }
}
```

Arama `search(0)` fonksiyonunu çağırarak başlar. Tahtanın büyüklüğü $n \times n$ idir ve kod toplam çözüm sayısını `count` değişkeninde tutar.

Kod bütün satır sütunların 0'dan $n-1$ 'e numaralandırıldığını varsayar. `search y` parametresi ile çağırıldığında `y`. satıra bir vezir koyar sonra kendisini `y + 1` parametresiyle çağırır. `y = n` olduğu zaman bir çözüm bulunmuş olur ve `count` 1 arttırılır.

`column` dizisi vezir içeren kolonları tutar. `diag1` ve `diag2` ise köşegenleri tutar. Örneğin 4×4 tahtada kolonlar ve köşegenler aşağıdaki gibidir:

0	1	2	3
0	1	2	3
0	1	2	3
0	1	2	3

column

0	1	2	3
1	2	3	4
2	3	4	5
3	4	5	6

diag1

3	4	5	6
2	3	4	5
1	2	3	4
0	1	2	3

diag2

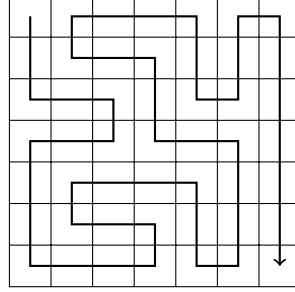
$q(n)$, $n \times n$ tahtada n tane veziri olası yerleştirme sayısını ifade etsin. Yukarıdaki geri izleme algoritması bize örneğin $q(8) = 92$ verir. n arttığında arama çok çabuk yavaşlar çünkü olası çözüm sayısı üstel (exponential) bir şekilde artar. Örneğin $q(16) = 14772512$ olur ve yukarıdaki algoritma ile sonucu bulmak modern bir bilgisayarda ortalama bir dakika sürer¹.

5.4 Aramayı Budama (Pruning The Search)

Arama ağacını budayarak geri izlemeyi optimize edebiliriz. Buradaki fikir algoritmaya "zeka" katarak yarı çözümü tam çözüme ulaşamayacağını en hızlı şekilde

¹ $q(n)$ ifadesinin büyük değerlerini bilinen hızlı bir şekilde bulmanın bir yolu yoktur. Şu anki rekor 2016'da hesaplanan $q(27) = 234907967154122528$ değeridir [55].

fark etmesidir. Bu tip optimizasyonlar aramanın verimliliğini ciddi derecede artırır. Diyelim $n \times n$ 'lik bir tahtada sol üst köşeden sağ alt köşeye her kareden sadece bir defa geçmek şartıyla oluşturulabilecek yol sayısını bulmaya çalışıyoruz. Örneğin 7×7 'lik bir tahtada sol üst köşeden sağ alt köşeye 111712 yol vardır. Bu yollardan biri aşağıdaki gibidir:



Zorluğu bizim ihtiyacımıza uygun olduğu için 7×7 'lik tahtada çalışıyoruz. İlk düz geri izleme algoritmasını kullanarak başlıyoruz ve sonrasında adım adım gözlemler yapılarak aramayı nasıl budayabileceğimizi buluyoruz. Her optimizasyondan sonra çalışma zamanını ve özyineleme çağrısını kaydediyoruz ki yapacağımız iyileştirmelerin etkisini aramamızda görebilelim.

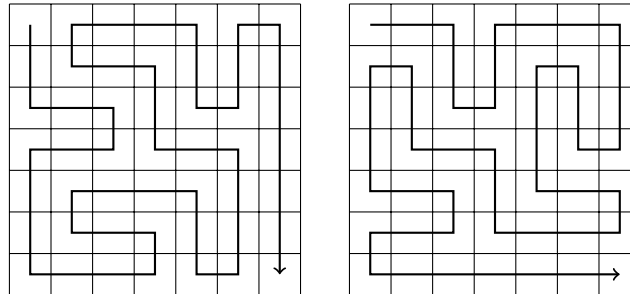
Düz Algoritma (Basic Algorithm)

Algoritmanın ilk versiyonu herhangi bir iyileştirme içermemektedir. Bütün olası sol yukarı köşeden sağ aşağı köşeye olan bütün olası yolları geri izleme yaparak sayıyoruz.

- çalışma süresi: 483 saniye
- özyineleme çağrı sayısı: 76 milyar

1. İyileştirme

Herhangi bir çözümde ilk başta ya sağ ya da aşağıya gidiyoruz. Her zaman tahtanın köşegeninden simetrik olan iğiki yol var. Örneğin aşağıdaki yollar birbirleriyle simetrik:

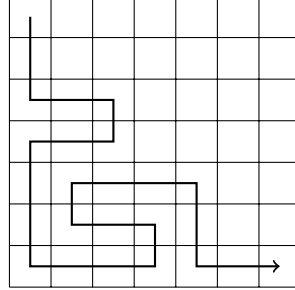


Bu yüzden hep ilk başta ya aşağı ya da sağa inip toplam çözüm sayısını ikiye çarpabiliriz.

- çalışma süresi: 244 saniye
- özyineleme çağrı sayısı: 38 milyar

2. İyileştirme

Eğer yol bütün hücreleri ziyaret etmeden sağ alt hücreye ulaşırsa bu çözüm mümkün olmayacaktır. Örnek olarak aşağıdaki yolu verebiliriz:

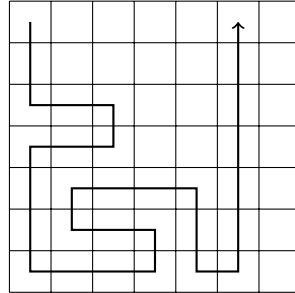


Bu gözlemi kullanarak eğer sağ alt hücreye erken vardıysak aramayı erkenden bitirebiliriz.

- çalışma süresi: 119 saniye
- özyineleme çağrı sayısı: 20 milyar

3. İyileştirme

Eğer yol bir sınıra dokunuyorsa ve sağa veya sola hareket edebiliyorsa tahta ziyaret edilmemiş hücreler içeren iki ayrı parçaya bölünür. Örneğin aşağıdaki durumda yol sağa veya sola dönebilir:

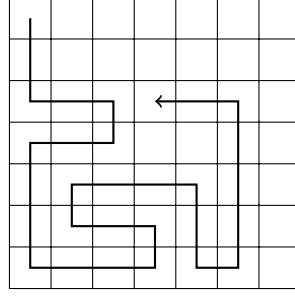


Bu durumda bütün hücreleri ziyaret edemeyiz ve aramayı erkenden bitirebiliriz. Bu iyileştirme bir hayli etkilidir:

- çalışma süresi: 1.8 saniye
- özyineleme çağrı sayısı: 221 milyon

4. İyileştirme

3. İyileştirme'deki fikir şu şekilde geliştirilebilir: Eğer yol ileri devam edemiyorsa ama sola veya sağa gidebilirse tahta ikisi de ziyaret edilmeyen hücreler içeren iki parçaya bölünür. Örneğin aşağıdaki yoldaki gibi:



Burada bütün hücreleri ziyaret edemeyeceğimiz için aramayı erkenden bitirebiliriz. Bu iyileştirmeden sonra aramamız gayet verimli hale geliyor:

- çalışma süresi: 0.6 saniye
- özyineleme çağrı sayısı: 69 milyon

Artık algoritmayı iyileştirmeyi bitirip şu ana kadar yaptığımıza bakabiliriz. Orijinal algoritma 483 saniyede çalışırken yaptığımız iyileştirmeler sayesinde çalışma süresi 0.6 saniyeye düştü. Böylece iyileştirmelerimiz sayesinde algoritma neredeyse 1000 kat daha hızlı oldu.

Geri izlemeye bu sıklıkla rastlanan bir durumdur çünkü arama ağacı genelde çok büyük olur ve çok basit gözlemleri kullanarak bu ağacı budamak etkili olabilir. Özellikle aramanın ilk adımlarında olan iyileştirmeler çok yararlı olur.

5.5 Ortada Buluşmak (Meet in the middle)

Ortada buluşmak arama uzayını eşit iki parçayı bölmek üzerine kurulmuştur. Her parça için ayrı bir arama yapılır ve sonda bu aramaların sonuçları birleştirilir.

bu teknik bu arama sonuçlarını hızlı bir şekilde birleştirme yolu varsa kullanılır. Böyle durumda iki arama tek bir aramadan daha az zaman gerektirebilir. Genelde, 2^n çarpanını ortada buluşmak yöntemiyle $2^{n/2}$ 'ye indirilebilir.

Örneğin n elemandan oluşan bir kümede toplamaları x olacak şekilde sayı seçmemiz isteniyor. Örneğin $[2, 4, 5, 9]$ kümesi ve $x = 15$ verilince çözüm olarak $[2, 4, 9]$ sayılarını $2 + 4 + 9 = 15$ olmasından dolayı alabiliriz. Fakat $x = 10$ mümkün değildir.

Bu problem için bütün alt kümelerle bakıp toplamaları x ediyor mu diye kontrol edebiliriz fakat bu algoritmanın çalışma süresi 2^n tane alt küme bulunmasından dolayı $O(2^n)$ olur. Fakat ortada buluşmak yöntemi ile $O(2^{n/2})$ çalışma süresine sahip daha bir algoritma kullanabiliriz².

$O(2^n)$ ve $O(2^{n/2})$ zaman karmaşıklıklarının farklı olduğuna dikkat ediniz çünkü $2^{n/2} \sqrt{2^n}$ 'e eşittir.

Buradaki fikir kümeyi sayıların yarısı bulunacak şekilde A ve B adında iki farklı kümeye bölmektir. İlk arama A kümesindeki bütün alt kümeleri bulur ve toplamalarını S_A dizisine kaydeder. Aynı şekilde ikinci arama B kümesinden S_B

²Bu algoritma, E. Horowitz ve S. Sahni tarafından 1974'te bulunmuştur [39].

dizisi oluşturur. Bundan sonra S_A 'dan ve S_B 'den bir eleman seçip toplamlarının x olup olmadığını kontrol edebiliriz. Bu, orijinal kümedeki sayılardan x toplamı oluşturabileceğimiz zaman mümkündür.

Örneğin $[2, 4, 5, 9]$ kümesine ve $x = 15$ sahip olduğumuzu düşünelim. İlk kümeyi $A = [2, 4]$ ve $B = [5, 9]$ diye iki kümeye böleriz. Sonra $S_A = [0, 2, 4, 6]$ ve $S_B = [0, 5, 9, 14]$ dizilerini oluştururuz. Bu durumda $x = 15$ 'in mümkün olabileceğini görürüz çünkü S_A , 6 içerir ve S_B , 9 içerir ve sonuç olarak $6 + 9 = 15$ olur. Böylece sonuç $[2, 4, 9]$ olur.

Bu algoritmayı $O(2^{n/2})$ zaman karmaşıklığına sahip olacak şekilde koda dökebiliriz. İlk *sırala* S_A ve S_B dizileri oluştururuz ki bu da birleştirme benzeri bir yöntemle $O(2^{n/2})$ zamanda yapılabilir. Diziler sıralandıktan sonra x toplamının S_A ve S_B dizilerinden yapılıp yapılamayacağına $O(2^{n/2})$ sürede bulabiliriz.

Bölüm 6

Açgözlü Algoritmalar (Greedy Algorithms)

A **Açgözlü algoritma** her zaman o anki en iyi gözüken kararı vererek yapılan çözümlere denir. Açgözlü algoritma asla kararlarını geri almaz ve direk son sonucu oluşturur. Bu yüzden genelde açgözlü algoritmalar verimlidir.

Açgözlü algoritma oluşturunun zorluğu her zaman en iyi cevabı verecek bir açgözlü stratejisi bulmaktır. Açgözlü algorithmadaki küçük optimal kararlara aynı zamanda genel olarak da optimal olmalıdır. Genelde bir açgözlü algoritmanın doğru çalışacağını kanıtlamak zordur.

6.1 Para Problemi (Coin problem)

İlk örnek, elimizdeki paraları kullanarak n miktarında para oluşturmamız isteniyor. Elimizdeki paraların değerleri $\text{coins} = \{c_1, c_2, \dots, c_k\}$ idir ve her parayı istediğimiz kadar kullanabiliriz. Toplam için gereken minimum kullanılması gereken para sayısı kaç tanedir?

Örneğin elimizde

$$\{1, 2, 5, 10, 20, 50, 100, 200\}$$

paraları olsun ve $n = 520$ oluşturmamız isteniyor. O zaman en az 4 tane paraya ihtiyaç duyarız ve optimal çözüm toplamı 520 olan $200 + 200 + 100 + 20$ olur.

Açgözlü Algoritma

Basit bir açgözlü algoritma, gereken miktar toplanana kadar her zaman seçilebilecek en büyük parayı seçmektir. Bu algoritma örnekteki durumda olur. Başta iki tan 200'lük para alırız ve sonrasında bir tane 100 ve bir tane 20'lik paralarla devam ederiz. Fakat bu algoritma her zaman doğru çözümü verir mi? Eğer elimizdeki paralar yukarıdaki paralar gibi olursa *her zaman* çalıştığını fark ederiz. Bu algoritmanın doğruluğu aşağıdaki yöntemle kanıtlanabilir:

1, 5, 10, 50 ve 100'lük paraların optimal çözümde hepsinden en fazla bir defa bulunabilir çünkü bu paralardan aynısı iki tane bulunursa bu paraları kaldırıp yerine bir tane parayla değiştirebiliriz. Örneğin $5 + 5$ içeriyorsa onları kaldırıp

yerine 10 koyabiliriz. Benzer mantıkta 2 ve 20'lik paralar çözümde en fazla iki defa bulunabilir çünkü $2+2+2$ paralarını $5+1$ ile $20+20+20$ paralarını ise $50+10$ ile daha optimal bir şekilde değiştirebiliriz. Hatta optimal bir çözüm $2+2+1$ veya $20+20+10$ içeremez çünkü bunları 5 ve 50 ile değiştirebiliriz.

Bu gözlemleri kullanarak her x paramız için x toplamını veya daha büyük toplamları sadece x 'ten küçük sayıları kullanarak yapamayız. Örneğin $x = 100$ ise sadece 100'den daha küçük paraları kullanarak yapabileceğimiz en optimal toplam $50+20+20+5+2+2 = 99$ olur. Bu örnek bize, algoritma kolay olsa dahi doğruluğunu kanıtlamanın zor olduğunu gösteriyor.

Genel Durum (General Case)

Genel durumda, sahip olduğumuz paralar rastgele paralar olabilir ve açgözlü algoritma bize her zaman optimal çözümü *vermeyebilir*.

Açgözlü algoritmanın her zaman doğru çalışmayacağını yanlış cevap verecek bir örnek vererek gösterebiliriz. Bu problemde rahatlıkla karşı bir örnek bulabiliriz:

Eğer elimizdeki paralar $\{1,3,4\}$ olursa ve istenen toplam 6 ise açgözlü algoritma $4+1+1$ çözümünü verir fakat en iyi çözüm $3+3$ olur.

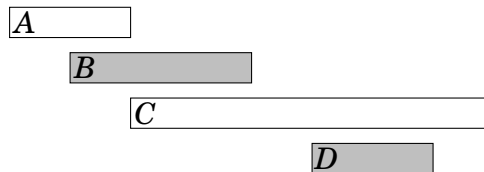
Para problemi için her zaman çalışan bir açgözlü algoritma bilinmemektedir¹. Fakat 7. Bölümde de göreceğimiz gibi bazı durumlarda genel problem her zaman doğru verecek şekilde dinamik programlama algoritması ile çözülebilir.

6.2 Zaman Planlaması (Scheduling)

Çoğu zaman planlama sorusu açgözlü algoritmalar ile çözülebilir. Klasik bir zaman planlama sorusu n tane başlangıç ve bitiş bilinen etkinlikten en çok etkinlik içerecek şekilde bir program oluşturmaktır. Burada etkinlikleri parça parça almayıp tam almanız gerekir. Örneğin aşağıdaki etkinliklere bakın:

etkinlik	başlama zamanı	bitiş zamanı
A	1	3
B	2	5
C	3	9
D	6	8

Bu durumda alınabilecek etkinlik sayısı ikidir. Örneğin **B** ve **D** etkinliklerini aşağıdaki gibi seçebiliriz:

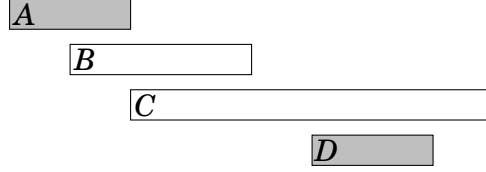


¹Fakat bu bölümde yapılan açgözlü algoritmanın elimizdeki paralar için doğru olup olmadığını polinom zamanda (polynomial time) kontrol etmek mümkündür [53].

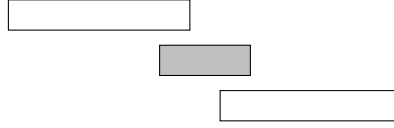
Bu soru için birkaç tane ağgözlü algoritma oluşturabiliriz ama hangileri her zaman çalışır?

1. Algoritma

İlk fikir her zaman *en küçük* etkinliği seçmektedir. Örneğin bu algoritma etkinlikleri aşağıdaki şekilde seçer.



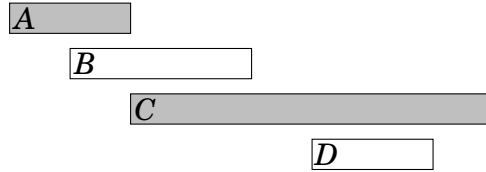
Fakat her zaman en kısa etkinlikleri almak doğru olmayabilir. Mesela algoritma aşağıdaki durumda hatalı cevap verir:;



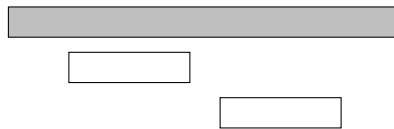
Eğer kısa etkinliği seçersek sadece bir etkinlik seçebiliyoruz fakat normalde uzun olan iki etkinliği de seçebiliyoruz.

2. Algoritma

Başka bir fikir ise her seferinde eklenebilecek en *erken başlayan* etkinliği eklemek. Bu algoritma aşağıdaki etkinlikleri seçer:



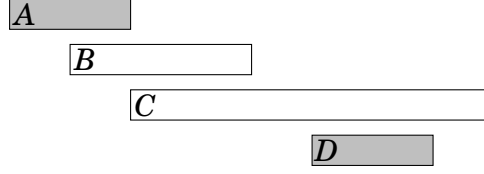
Fakat bu algoritmaya da karşı bir örnek bulabiliriz. Örneğin aşağıdaki durumda bu algoritma sadece bir tane etkinlik seçer.



Eğer ilk etkinliği seçersek diğer etkinlikleri seçebilmemiz mümkün değildir fakat diğer durumda diğer iki tane etkinliği seçebiliriz.

3. Algoritma

3. fikir ise her seferinde seçilebilecek en *erken biten* etkinliği seçmektir. Bu algoritma aşağıdaki etkinlikleri seçer:



Bu algoritmanın *her zaman* doğru cevap verdiğini görüyoruz. Bunun sebebi ise ilk başta en erken biten etkinliği seçmenin optimal olmasıdır. Bundan sonra diğer etkinlikler için de aynısını uygulayabiliriz.

Algoritmanın çalıştığını kanıtlama yollarından biri de ilk başta erken biten bir etkinlik yerine daha geç biten bir etkinlik seçince ne olduğuna bakmaktır. Sonuç olarak seçebileceğimiz etkinlik sayısı en iyi durumda öncekiyle eşit olabilir. Bu yüzden daha geç biten bir etkinliği seçmek hiçbir zaman daha iyi bir sonuç vermez ve bu yüzden bu açgözlü algoritma doğrudur.

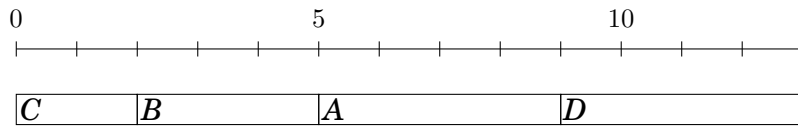
6.3 Görevler ve Son Teslimler (Tasks and Deadlines)

Süresi ve son teslimlerini bildiğimiz n tane görevimiz olduğunu düşünelim. Amacımız görevleri yapmak için bir sıra oluşturmak. Her görev için $d - x$ puan kazanıyoruz. d görevin son teslimi ve x ise görevi bitirdiğimiz zaman oluyor. En fazla alabileceğimiz toplam puan kaçtır?

Örneğin görevlerin aşağıdaki gibi olduğunu düşünelim:

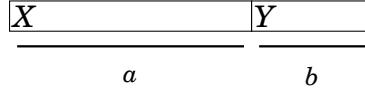
görev	süre	son teslim
A	4	2
B	3	5
C	2	7
D	4	5

Bu durumda en iyi sıralardan biri aşağıdaki gibi olur:

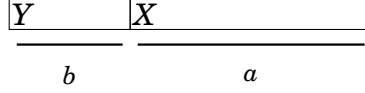


Bu çözümde C 5 puan, B 0 puan, A -7 puan ve D ise -8 puan verip toplam puan -10 olur.

İlginç bir şekilde sorunun optimal çözümü son teslimlere bağlı olmuyor çünkü doğru bir açgözlü algoritma görevleri *sürelerine göre artan bir şekilde* sıralamak oluyor. Bunun nedeni ise eğer daha uzun süren bir görevi daha önce yaparsak bu görevlerin yerlerini değiştirirsek daha iyi bir çözüm elde etmiş oluruz. Örneğin aşağıdaki sıraya bakın:



Burada $a > b$ olduğu için görevlerin yerlerini değiştirmemiz gerekir.



Şimdi X , b puan az verirken Y a puan fazla verir ve toplam puan $a - b > 0$ kadar artar. Optimal bir çözümde ardışık iki görevde kısa olan görevin uzun olandan daha önce yapılması gerekir. Bu yüzden görevler yapılma sürelerine göre kısıdan uzuna doğru sıralanır.

6.4 Toplamları Küçültmek (Minimizing Sums)

Diyelim ki bize n tane a_1, a_2, \dots, a_n sayıları verildiğini düşünelim amacımız öyle bir x bulmaktır ki bu değer

$$|a_1 - x|^c + |a_2 - x|^c + \dots + |a_n - x|^c.$$

toplamını en küçük yapacaktır. $c = 1$ ve $c = 2$ olduğu durumları inceleyeceğiz.

$c = 1$ Durumu

Bu durumda

$$|a_1 - x| + |a_2 - x| + \dots + |a_n - x|.$$

toplamını en küçük hale getirmemiz gerekir. Örneğin sayılarımız $[1, 2, 9, 2, 6]$ olunca en iyi çözümü $x = 2$ verir. Bu değer sonuç olarak

$$|1 - 2| + |2 - 2| + |9 - 2| + |2 - 2| + |6 - 2| = 12$$

verir. Genel durumda en iyi sonucu veren x değeri sayıların *medyanı* olur. Medyan sayıları sıraladıktan sonra ortadaki sayıdır (Çift miktarda sayı var ise ortadaki iki sayının ortalamasıdır.). Örneğin $[1, 2, 9, 2, 6]$ dizisi sıralandıktan sonra $[1, 2, 2, 6, 9]$ olur ve medyanı 2'dir.

Medyan en iyi çözümlerden biridir çünkü x medyandan küçük ise toplam x 'i arttırarak toplam azalır ve x medyandan büyükse x 'i azaltarak toplamı küçültebiliriz. Eğer n çift sayı ise ve iki tane medyan varsa iki medyan da ve arasındaki bütün değerler optimal çözümlerdir.

$c = 2$ Durumu

Bu durumda

$$(a_1 - x)^2 + (a_2 - x)^2 + \dots + (a_n - x)^2.$$

toplamını küçültmemiz gerekir. Örneğin sayılar $[1, 2, 9, 2, 6]$ ise en iyi çözüm $x = 4$ ile gelir. $x = 4$

$$(1 - 4)^2 + (2 - 4)^2 + (9 - 4)^2 + (2 - 4)^2 + (6 - 4)^2 = 46.$$

çözümünü verir. Genel durumda en iyi x değeri sayıların *ortalamasıdır*. Örneğin yukarıdaki sayıların ortalaması $(1 + 2 + 9 + 2 + 6)/5 = 4$ idir ve bu sonuç, toplamı aşağıdaki hale getirilerek gösterilebilir:

$$nx^2 - 2x(a_1 + a_2 + \dots + a_n) + (a_1^2 + a_2^2 + \dots + a_n^2)$$

Son kısım x 'e bağlı olmadığı için umursamayabiliriz ve geri kalan kısımlar bir $nx^2 - 2xs$ fonksiyonu oluşturur. Burada $s = a_1 + a_2 + \dots + a_n$ idir. Bu durum kökleri $x = 0$ ve $x = 2s/n$ olan ve yukarıya doğru açılan bir parabol (concave up) oluşturur. En küçük değer köklerin ortalaması yani $x = s/n$ idir yani a_1, a_2, \dots, a_n sayılarının ortalamasıdır.

6.5 Veri Sıkıştırma (Data Compression)

Bir **binary kodu** bir yazının her karakteri için bitlerden oluşan bir **şifre** oluşturur. Bu yazıyı binary kodu kullanarak her harfı uygun şifre ile değiştirerek *sıkıştırabiliriz*. Örneğin aşağıdaki binary kodu A-D karakterlerini şu şekilde şifreler:

karakter	şifre
A	00
B	01
C	10
D	11

Bu **sabit uzunlukta** bir koddur yani her şifrenin uzunluğu aynıdır. Örneğin AABACDACA yazısını aşağıdaki gibi sıkıştırabiliriz:

000001001011001000

Bu kodu kullanarak sıkıştırılmış yazı 18 bit olur. Fakat bu yazıyı **değişken uzunluklu** kod ile daha iyi bir şekilde sıkıştırabiliriz. Bu durumda karakter şifreleri farklı uzunlukta olabilir. Sonrasında çok bulunan karakterler için kısa şifreler verirken az bulunan karakterler içinse daha uzun şifreler veririz. Bu durumda yukarıdaki yazı için en iyi kod aşağıdaki gibi olur:

karakter	şifre
A	0
B	110
C	10
D	111

Optimal kod sıkıştırılmış bir yazıyı olabildiğince küçük hale getirmeye çalışır. Bu durumda optimal kod

001100101110100,

olup gereken bit sayısı 18 yerine 15'tir. Böylece daha iyi bir kod sayesinde 3 bit kurtarmış oluruz.

Bunu uygulayabilmemiz için herhangi bir şifrenin başka bir şifrenin ön eki(prefix) olmamalıdır. Örneğin bir kodda hem 10 hem de 1011 şifrelerine aynı anda izin verilmemektedir. Bunun nedeni ise sıkıştırdığımız yazıdan orijinal yazıyı alabilmeyi istememizdir. Eğer bir şifre başka bir şifrenin ön eki olursa bu durum mümkün olmaz. Örneğin aşağıdaki kod uygun *değildir*:

karakter	şifre
A	10
B	11
C	1011
D	111

Bu kodu kullanarak 1011 yazısının AB mi olduğunu yoksa C mi olduğunu anlayamayız.

Huffman Kodlaması (Huffman Coding)

Huffman Kodlaması² bir yazıyı sıkıştırmak için en optimal kodu veren bir açgözlü algoritmadır. Algoritma ikili ağaç (binary tree) oluşturur ve bu ağaç karakterlerin kelimedeki bulunma sıklıklarına göre şekillenir. Her karakterin şifresi kökten istenen karakterin düğümüne gidilerken bulunabilir. Ağaçta sola ilerlemek 0 biti eklemek, sağa ilerlemek ise 1 biti eklemek anlamına gelir.

Başta yazının her karakteri bir düğümle gösterilir. Bu düğümlerin ağırlıkları yazıda düğümdeki karakterin kaç defa bulunduklarına eşittir. Her adımda en küçük iki ağırlığa sahip düğüm birleştirilir ve ağırlığı bu iki ağırlığın toplamı olan yeni bir düğüm oluşturulur. Bu işlem her düğüm birleşene kadar devam eder.

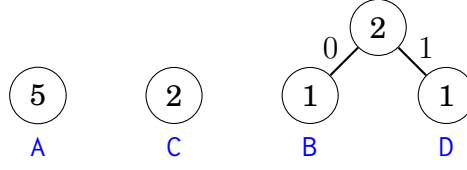
Şimdi Huffman Kodlamasının AABACDACA için en iyi kodu oluşturmasını göreceğiz. Başta yazıdaki karakterlere denk gelen dört tane düğüm var.



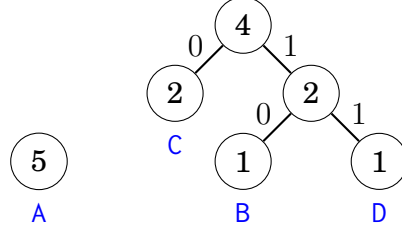
A karakterini gösteren düğümün ağırlığı 5'tir çünkü A karakteri yazıda 5 defa bulunmaktadır. Diğer ağırlıklar da aynı şekilde hesaplanmıştır.

İlk adım B ve D karakterlerini gösteren düğümleri birleştirmek çünkü ikisinin de ağırlığı en küçük olup 1'dir. Sonuç:

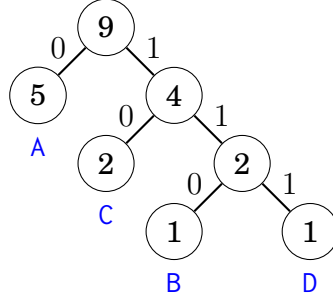
²D. A. Huffman bu methodu üniversite dersinde bir soruyu çözerken bulmuştur ve bu algoritmayı 1952'de yayınlamıştır [40].



Bundan sonra 2 ağırlığındaki düğümler birleştiriliyor:



En sonda kalan iki düğüm birleştiriliyor:



Şimdi bütün düğümler ağaçta bulunduğu için kod hazır durumdadır. Ağaçtan aşağıdaki şifreler bulunabilir:

karakter	şifre
A	0
B	110
C	10
D	111

Bölüm 7

Dinamik Programlama

Dinamik programlama tam bir aramanın doğruluğu ile aç gözlü algoritmaların verimliliğinin birleştiren bir tekniğdir. Eğer problemde alt problemler birkaç defa çözülüyorsa ve bu alt problemlere bağımsız bir şekilde çözebiliyorsak dinamik programlama kullanabiliriz.

Dinamik programlamanın iki kullanımı vardır:

- **Optimal bir çözüm bulmak:** Olabildiğince büyük veya olabildiğince küçük bir çözüm istiyoruz.
- **Olası çözüm sayısını hesaplama:** Toplam olası çözüm sayısını hesaplamak istiyoruz.

İlk dinamik programlamanın optimal çözüm bulmakta nasıl kullanıldığına bakacağız ve sonra aynı fikri çözüm sayarken kullanacağız.

Dinamik programlamayı anlamak her rekabetçi programlamacı / olimpiyatçı için bir dönüm noktasıdır. Basit fikri kolay olsa da asıl zorluk dinamik programlamayı farklı problemlere uygulamaktır. Bu bölüm iyi bir temel olması için bazı klasik soruları gösterecek.

7.1 Para Problemi

İlk başta Bölüm 6'da gördüğümüz bir problem üzerine yoğunlaşacağız: $\text{coins} = \{c_1, c_2, \dots, c_k\}$ değerlerin oluşan bir para kümesinde amacımız toplamı hedef toplamı n 'i oluşturan ve en az sayıda para kullandığımız para kümesini bulmaktır.

Bölüm 6'da bu problemi her zaman mümkün en büyük parayı kullanarak çözmüştük. Bu aç gözlü algoritma örneğin Euro paralarında çalışır fakat genel durumda bu aç gözlü algoritma her zaman optimal çözümü vermez.

Şimdi problemi verimli bir şekilde dinamik programlama kullanarak çözelim ki böylece algoritma her para kümesi üzerinde çalışabilsin. Dinamik programlama aynı bir kaba kuvvet algoritması gibi toplamı oluşturabilecek bütün olasılıklardan özyinelemeli bir fonksiyon ile geçer. Fakat dinamik programlama verimlidir çünkü *memoization* kullanır yani önceden hesaplanan değerleri hatırlar ve böylece her alt problemi en fazla bir defa çözeriz.

Özyinelemeli Formülleştirme

Dinamik programlamada fikir problemi özyinelemeli bir şekilde formüle edip problemin çözümünü daha küçük alt problemlerin çözümlerinden bulunabilir hale getirmektir. Para probleminde doğal bir özyineleme problemi şöyledir: Toplamları x toplamını oluşturan en az para sayısı kaçtır?

$\text{solve}(x)$, x toplamı için gerekli olan minimum para sayısını belirtsin. Fonksiyonun değerleri paraların değerlerine bağlıdır. Örneğin $\text{coins} = \{1, 3, 4\}$ ise fonksiyonun ilk değerleri aşağıdaki gibidir:

$\text{solve}(0)$	$=$	0
$\text{solve}(1)$	$=$	1
$\text{solve}(2)$	$=$	2
$\text{solve}(3)$	$=$	1
$\text{solve}(4)$	$=$	1
$\text{solve}(5)$	$=$	2
$\text{solve}(6)$	$=$	2
$\text{solve}(7)$	$=$	2
$\text{solve}(8)$	$=$	2
$\text{solve}(9)$	$=$	3
$\text{solve}(10)$	$=$	3

Örneğin $\text{solve}(10) = 3$ çünkü 10 toplamını oluşturmak için en az 3 para gerekir. Optimal çözüm ise $3 + 3 + 4 = 10$ olur.

solve fonksiyonunun önemli bir özelliği değerlerinin özyinelemeli bir şekilde daha küçük değerlerden hesaplanabiliyor olmasıdır. Buradaki fikir toplam için *ilk* seçtiğimiz paraya odaklanmaktır. Örneğin yukarıdaki durumda ilk para 1, 3 veya 4 olabilir. Eğer ilk 1 parasını seçersek şimdiki görevimizi 9 toplamını en az sayıda para toplayarak bulmaktır ki bu da orijinal sorunun bir alt problemidir. Tabii aynı durum 3 ve 4 paraları için de geçerlidir. Böylece özyineleme formülünü en az sayıda parayı hesaplamak için bulabiliriz:

$$\begin{aligned}\text{solve}(x) = \min(&\text{solve}(x - 1) + 1, \\ &\text{solve}(x - 3) + 1, \\ &\text{solve}(x - 4) + 1).\end{aligned}$$

Özyinelemenin temel durumu $\text{solve}(0) = 0$ olur çünkü boş toplam hesaplamak için hiç paraya ihtiyaç duyulmaz. Örneğin

$$\text{solve}(10) = \text{solve}(7) + 1 = \text{solve}(4) + 2 = \text{solve}(0) + 3 = 3.$$

Şimdi x toplamını veren en az miktarda para sayısını hesaplayan bir genel özyineleme fonksiyonu yazabiliriz:

$$\text{solve}(x) = \begin{cases} \infty & x < 0 \\ 0 & x = 0 \\ \min_{c \in \text{coins}} \text{solve}(x - c) + 1 & x > 0 \end{cases}$$

İlk, $x < 0$ ise değer ∞ olur çünkü negatif miktarda bir para hesaplamak imkansızdır. Eğer $x = 0$ ise değer 0 olur çünkü boş toplam oluşturmak için paraya ihtiyaç yoktur. Eğer $x > 0$ ise c değişkeni toplamın ilk parasını seçmek için bütün olasılıklardan geçer.

Problemi çözen bir özyinelemeli fonksiyon bulunduğu zaman direk çözümü C++'da yazabiliriz (INF sabiti sonsuzu ifade eder.):

```
int solve(int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    int best = INF;
    for (auto c : coins) {
        best = min(best, solve(x-c)+1);
    }
    return best;
}
```

Yine de fonksiyon verimli değildir çünkü toplamı oluşturmak için üstel (çok) miktarda yol vardır. Fakat şimdi fonksiyonu memoization ile nasıl verimli yapabileceğimiz göreceğiz.

Memoization

Dinamik programlamadaki fikir özyinelemeli bir fonksiyonun değerini verimli bir şekilde hesaplamak için **memoization** kullanmaktır. Bu, fonksiyondaki değerlerin hesaplandıktan sonra bir dizide tutulacağı anlamına gelir. Her parametre için fonksiyonun değeri özyinelemeli şekilde en fazla bir defa hesaplanır ve bundan sonra değer direkt diziden alınabilir.

Problemde

```
bool ready[N];
int value[N];
```

`ready[x]` ifadesi `solve(x)` değerinin hesaplanıp hesaplanmadığını belirtir ve eğer hesaplandıysa `value[x]` bu değeri tutar. N sabiti dizide gereken bütün değerleri tutmak için seçilmiştir.

Şimdi fonksiyon verimli bir şekilde aşağıdaki gibi hesaplanabilir:

```
int solve(int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    if (ready[x]) return value[x];
    int best = INF;
    for (auto c : coins) {
        best = min(best, solve(x-c)+1);
    }
    value[x] = best;
    ready[x] = true;
}
```

```
    return best;
}
```

Fonksiyon önce $x < 0$ ve $x = 0$ durumlarına bakıyor. Sonra fonksiyon `ready[x]` ifadesini kontrol ederek `solve(x)` fonksiyonunun zaten önceden hesaplanıp hesaplanmadığına bakar. Eğer hesaplandıysa fonksiyon `value[x]` değerini döndürür ve eğer hesaplanmadıysa da `solve(x)` değerini özyinelemeli bir şekilde hesaplar ve `value[x]`'de tutar.

Fonksiyon verimli bir şekilde çalışır çünkü her x parametresi özyinelemeli şekilde bir defa hesaplanır. Bir `solve(x)` fonksiyonunun değeri `value[x]`'de hesaplandıysa, verimli bir şekilde fonksiyonun bir daha x parametresiyle çağrılıp çağrılmayacağı bulunabilir. Bu algoritmanın zaman karmaşıklığı $O(nk)$ olur ve n istenen toplamken k de toplam para sayısıdır.

Bu arada `value` dizisini özyineleme (recursive) yerine *döngüyle* (iteratively) hesaplayabiliriz. Bu döngü basit şekilde $0 \dots n$ parametreleri için bütün `solve` değerlerini hesaplar:

```
value[0] = 0;
for (int x = 1; x <= n; x++) {
    value[x] = INF;
    for (auto c : coins) {
        if (x-c >= 0) {
            value[x] = min(value[x], value[x-c]+1);
        }
    }
}
```

Hatta çoğu rekabetçi programlamacı bu implementasyonu tercih eder çünkü bu daha kısadır ve daha küçük sabit faktörleri vardır. Bundan sonra örneklerimizde aynı zamanda döngü implementasyonlarını kullanacağız. Yine de dinamik programlama çözümlerini özyinelemeli fonksiyon ile düşünmek daha kolay olur.

Çözüm Oluşturmak

Bazen bir optimal çözümün değerini bulmamız istenirken bu çözümü nasıl oluşturacağımızı da bulmamızı isterler. Para probleminde örneğin optimal çözümde her para toplamı için örneğin kullanacağımız ilk parayı gösteren bir dizi oluşturabiliriz:

```
int first[N];
```

Sonra algoritmayı aşağıdaki şekilde değiştiririz:

```
value[0] = 0;
for (int x = 1; x <= n; x++) {
    value[x] = INF;
    for (auto c : coins) {
        if (x-c >= 0 && value[x-c]+1 < value[x]) {
```

```

        value[x] = value[x-c]+1;
        first[x] = c;
    }
}
}

```

Bundan sonra aşağıdaki kod n toplamının optimal bir çözümünde bulunan parametreleri yazdırır:

```

while (n > 0) {
    cout << first[n] << "\n";
    n -= first[n];
}

```

Çözüm Sayısını Hesaplamak

Şimdiki amacımız x toplamını veren para kümesi sayısını bulmaktır. Örneğin $\text{coins} = \{1, 3, 4\}$ ve $x = 5$ ise toplam 6 yol vardır:

- $1 + 1 + 1 + 1 + 1$
- $1 + 1 + 3$
- $1 + 3 + 1$
- $3 + 1 + 1$
- $1 + 4$
- $4 + 1$

Yine bu problemi özyinelemeli bir şekilde çözebiliriz. $\text{solve}(x)$ x toplamını veren kümelerin sayısını versin. Örneğin $\text{coins} = \{1, 3, 4\}$ ise $\text{solve}(5) = 6$ olur ve özyinelemeli formül

$$\begin{aligned} \text{solve}(x) = & \text{solve}(x-1) + \\ & \text{solve}(x-3) + \\ & \text{solve}(x-4). \end{aligned}$$

şeklinde olur. Sonra genel özyineleme formülü aşağıdaki gibi olur:

$$\text{solve}(x) = \begin{cases} 0 & x < 0 \\ 1 & x = 0 \\ \sum_{c \in \text{coins}} \text{solve}(x-c) & x > 0 \end{cases}$$

Eğer $x < 0$ ise değer 0 olur çünkü hiç çözüm bulunmaz ve eğer $x = 0$ ise değer 1'dir çünkü boş toplam oluşturma'nın bir yolu vardır (hiç sayı almamak). Diğer türlü c 'nin coins 'de bulunan bir para olduğu düşünülerek bütün $\text{solve}(x-c)$ formundaki değerlerin toplamını hesaplarız.

Aşağıdaki kod bir `count` dizisi oluşturur ki `count[x]`, $0 \leq x \leq n$ parametreleri için $\text{solve}(x)$ değerine eşit olur:

```

count[0] = 1;
for (int x = 1; x <= n; x++) {

```

```

for (auto c : coins) {
    if (x-c >= 0) {
        count[x] += count[x-c];
    }
}
}

```

Genelde toplam çözüm sayısı çok büyük olduğu için kesin sayıyı bulmamız istenmez fakat çözümün mod m 'de bulmamız istenir ve örneğin $m = 10^9 + 7$ olur. Bu, yukarıdaki kodu bütün işlemlerin mod m 'de yapılması sağlanarak sağlanabilir. Yukarıdaki kodda

```
count[x] += count[x-c];
```

satırından sonra

```
count[x] %= m;
```

ifadesini eklemek yeterlidir.

Şimdiye kadar bütün basit dinamik programlama fikirlerinden bahsettik. Dinamik programlama çeşitli durumlarda kullanılabileceği için şimdi dinamik programın başka farklı olasıklarını gösteren bazı problemlere bakacağız.


7.2 En Uzun Artan Altdizi (Longest Increasing Subsequence)

İlk problememiz n elemandan oluşan bir dizide **en uzun artan altdizi**yi bulmaktır. Bu soldan sağa doğru gidip alt dizideki her elemanın önceki elemandan büyük olduğu maksimum uzunlukta bir altdizidir. Elemanlar ardışık sırada bulunmak zorunda değildir. Örneğin

0	1	2	3	4	5	6	7
6	2	5	1	7	4	8	3

dizisinde en uzun artan altdizi 4 eleman içerir:

0	1	2	3	4	5	6	7
6	2	5	1	7	4	8	3



$\text{length}(k)$, k pozisyonunda biten en uzun artan altdizi uzunluğunu belirtir. Böylece $0 \leq k \leq n - 1$ olduğu bütün $\text{length}(k)$ değerlerini hesaplırsak en uzun artan alt dizinin uzunluğunu bulacağız. Örneğin yukarıdaki dizinin değerleri aşağı-

ğdaki gibidir:

```
length(0) = 1
length(1) = 1
length(2) = 2
length(3) = 1
length(4) = 3
length(5) = 2
length(6) = 4
length(7) = 2
```

Örneğin $\text{length}(6) = 4$ olur çünkü 6. pozisyonda biten en uzun artan alt-dizi 4 eleman içerir. $\text{length}(k)$ değerini hesaplamak için $\text{array}[i] < \text{array}[k]$ olup $\text{length}(i)$ değeri olabildiğince büyük olan bir $i < k$ pozisyonu bulmak gerekir. Sonra $\text{length}(k) = \text{length}(i) + 1$ olduğunu biliyoruz çünkü bu $\text{array}[k]$ bir alt-diziye eklemek için optimal bir yoldur. Fakat eğer böyle bir i pozisyonu yoksa $\text{length}(k) = 1$ olur çünkü alt-dizi $\text{array}[k]$ sayısını içerir.

Fonksiyondaki bütün değerler daha değerlerden hesaplanabildiği için dinamik programlama kullanabiliriz. Yukarıdaki kodda, fonksiyonun değerleri bir `length` dizisinde tutulur.

```
for (int k = 0; k < n; k++) {
    length[k] = 1;
    for (int i = 0; i < k; i++) {
        if (array[i] < array[k]) {
            length[k] = max(length[k], length[i]+1);
        }
    }
}
```

Bu kod $O(n^2)$ zamanda çalışır çünkü iki tane iç içe döngüden oluşur. Fakat dinamik programlama hesaplamasını daha verimli bir şekilde $O(n \log n)$ zamanda implement edebiliriz. Bunu yapan bir yol bulabilir misin?

7.3 Düzlemde Yollar

Şimdiki problemimiz $n \times n$ hücreden oluşan bir düzlemde sol üst köşeden sağ alt köşeye giden bir yol bulmaktır. Her kare pozitif bir sayı içerir ve her yol, o yol üzerindeki değerlerin toplamı o hücreye oluşturulabilecek yolların değerleri arasında en büyükse oluşturulabilir.

Aşağıdaki resim düzlemde optimal bir yol gösterir:

3	7	9	2	7
9	8	3	5	5
1	7	9	8	5
3	8	6	4	10
6	3	9	7	8

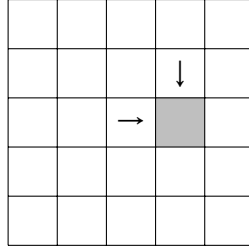
Yol üzerisindeki değerlerin toplamı 67 olur ve bu sol üst köşeden sağ alt köşeye giden en büyük toplama sahip yoldur.

Diyelim ki satırlar ve sütunlar 1'den n 'e numaralandırılmıştır ve $value[y][x]$ değeri (y, x) karesine eşittir. $sum(y, x)$ sol üst köşeden (y, x) karesine giden bir yolun en büyük toplamını gösterir. Şimdi $sum(n, n)$, bize sol üst köşeden sağ alt köşeye en büyük toplamı verir. Örneğin yukarıdaki düzlemde $sum(5, 5) = 67$.

Toplamı özyinelemeli bir şekilde aşağıdaki gibi hesaplayabiliriz:

$$sum(y, x) = \max(sum(y, x - 1), sum(y - 1, x)) + value[y][x]$$

Özyinelemeli formül, (y, x) karesinde biten bir yolun ya $(y, x - 1)$ karesinden ya da $(y - 1, x)$ karesinden geldiği gözlemine dayanır:



Böylece toplamı en büyük yapan yönü seçeriz. Eğer $y = 0$ veys $x = 0$ ise $sum(y, x) = 0$ olduğunu varsayabiliriz çünkü bunu sağlayan herhangi bir yol yoktur. Böylece özyinelemeli formül ya $y = 1$ iken ya da $x = 1$ iken çalışır.

sum fonksiyonunun iki parametresi olduğu için dinamik programlama dizisinin de iki tane boyutu vardır. Örneğin

```
int sum[N][N];
```

dizisini kullanıp toplamı aşağıdaki şekilde buluruz:

```
for (int y = 1; y <= n; y++) {
    for (int x = 1; x <= n; x++) {
        sum[y][x] = max(sum[y][x-1], sum[y-1][x]) + value[y][x];
    }
}
```

Algoritmanın zaman karmaşıklığı $O(n^2)$ olur.

7.4 Knapsack Problemleri

Knapsack terimi bir obje kümesinin verilip bazı özelliklere sahip alt kümeleri buldurmayı gerektiren sorulara denir. Knapsack problemleri genelde dinamik programlama ile çözülebilir.

Bu bölümde şu problemle ilgileneceğiz: $[w_1, w_2, \dots, w_n]$ ağırlıklarının verildiği bilinen bu ağırlıklarla oluşturulabilecek bütün toplam ağırlıkları bulmak. Örneğin ağırlıklar $[1, 3, 3, 5]$ ise aşağıdaki toplamalar mümkündür:

0	1	2	3	4	5	6	7	8	9	10	11	12
X	X		X	X	X	X	X	X	X		X	X

Bu durumda 2 ve 10 hariç geriye kalan bütün $0 \dots 12$ arasındaki sayılar mümkündür. Örneğin 7 toplamı mümkündür çünkü $[1,3,3]$ ağırlıklarını seçebiliriz.

Problemi çözmek için toplam oluşturmak sadece ilk k ağırlığı kullanabildiğimiz alt problemlere odaklanılır. Eğer ilk k ağırlıktan bazı ağırlıkları kullanarak x toplamı elde edebiliyorsak $\text{possible}(x,k) = \text{true}$ doğru olur ve aksi taktirde $\text{possible}(x,k) = \text{false}$ olur. Fonksiyonun değerleri özyinelemeli bir şekilde aşağıdaki gibi hesaplanabilir:

$$\text{possible}(x,k) = \text{possible}(x-w_k, k-1) \vee \text{possible}(x, k-1)$$

Formül, toplamda w_k ağırlığını isteğimize göre kullanıp kullanmayacağımıza bağlıdır. Eğer w_k kullanırsak geriye kalan göre ilk $k-1$ ağırlıktan $x-w_k$ toplamı oluşturmaktır ve eğer w_k 'i kullanmazsak geri kalan görevimiz ilk $k-1$ ağırlıktan x toplam oluşturmaktır. Temel durumlar

$$\text{possible}(x,0) = \begin{cases} \text{true} & x = 0 \\ \text{false} & x \neq 0 \end{cases}$$

olur çünkü hiç ağırlık kullanmazsak sadece 0 toplamını oluşturabiliriz.

Aşağıdaki tablo $[1,3,3,5]$ ağırlıkları için fonksiyonun değerlerini gösterir ("X" semboolü doğru değerleri gösterir.).

$k \backslash x$	0	1	2	3	4	5	6	7	8	9	10	11	12
0	X												
1	X	X											
2	X	X		X	X								
3	X	X		X	X		X	X					
4	X	X		X	X	X	X	X	X	X		X	X

Bu değerleri hesapladıktan sonra $\text{possible}(x,n)$ bize *bütün* ağırlıkları kullanarak x toplamını oluşturup oluşturamayacağımızı verir.

W , bütün ağırlıkların toplamını belirtsin. O zaman $O(nW)$ zamanda çalışan dinamik programlama çözümü aşağıdaki özyinelemeli fonksiyona karşılık gelir:

```
possible[0][0] = true;
for (int k = 1; k <= n; k++) {
    for (int x = 0; x <= W; x++) {
        if (x-w[k] >= 0) possible[x][k] |= possible[x-w[k]][k-1];
        possible[x][k] |= possible[x][k-1];
    }
}
```

Fakat sadece tek boyutlu `possible` dizisini kullanan daha iyi bir implemantasyon vardır. `possible[x]` ifadesi x toplamını sağlayan bir alt küme oluşturup oluşturamayacağımızı gösterir. Buradaki mantık, diziyi sağdan sola doğru her yeni ağırlıkta diziyi güncellemektir:

```
possible[0] = true;
for (int k = 1; k <= n; k++) {
    for (int x = W; x >= 0; x--) {
        if (possible[x]) possible[x+w[k]] = true;
    }
}
```

Buradaki genel fikir çoğu knapsack probleminde kullanılabilir. Örneğin eğer ağırlıklar ile değerler verildiyse her ağırlık toplamı için maksimum değeri de bulabiliriz.

7.5 Değişiklik Farkı (Edit Distance)

Değişiklik farkı veta **Levenshtein farkı**¹ bir yazıyı(string) başka bir yazıya dönüştürmek için gereken minimum değişiklik sayısını söyler. Yapabileceğimiz değişiklikler aşağıdaki gibidir:

- bir karakter ekle (e.g. ABC → ABCA)
- bir karakter çıkar (e.g. ABC → AC)
- bir karakter değiştir (e.g. ABC → ADC)

Örneğin LOVE ve MOVIE arasındaki mesafe 2'dir çünkü ilk başta LOVE → MOVE (değiştirme) yapıp sonra MOVE → MOVIE (ekleme) yapabiliriz. Bu olası en az operasyon gerektiren durumdur çünkü bir operasyonun yeterli olmadığı bariz bir şekilde görünmektedir.

Diyelim ki bize bir n uzunluğunda bir x yazısı ile m uzunluğunda bir y yazısı verildiğini düşünelim. Amacımız x ve y arasındaki değişiklik farkını hesaplamak olsun. Bu problemi çözmek için `distance` fonksiyonunu kullanırız. `distance(a,b)`, $x[0...a]$ ile $y[0...b]$ ön ekleri arasındaki değişiklik farkını tutar. Böylece bu fonksiyonu kullanarak x ile y arasındaki değişiklik farkı `distance(n-1,m-1)` olur.

`distance`'ın değerlerini aşağıdaki gibi hesaplayabiliriz:

$$\begin{aligned} \text{distance}(a,b) = \min(&\text{distance}(a,b-1) + 1, \\ &\text{distance}(a-1,b) + 1, \\ &\text{distance}(a-1,b-1) + \text{cost}(a,b)). \end{aligned}$$

Burada $x[a] = y[b]$ ise $\text{cost}(a,b) = 0$ yoksa $\text{cost}(a,b) = 1$ olur. Formül, x yazısını değiştirmek için aşağıdaki yolları düşünür:

- `distance(a,b-1)`: x 'in sonuna bir karakter ekle.
- `distance(a-1,b)`: x 'in sonundan bir karakter çıkar.
- `distance(a-1,b-1)`: x 'in son karakterini eşle veya değiştir.

¹Bu kavram V. I. Levenshtein'dan ismini almıştır. Kendisi ikili kodlarla bağlantıyı araştırmıştır [49].

İlk iki durumda sadece bir değişiklik gerekir (ekleme veya çıkarma). Son durumda $x[a] = y[b]$ ise son karakterleri değiştirmeden eşleyebiliriz yoksa bir değişiklik operasyonu gerekir (değiştirme).

Aşağıdaki tablo örnek durumdaki distance değerlerini gösterir:

		M	O	V	I	E
	0	1	2	3	4	5
L	1	1	2	3	4	5
O	2	2	1	2	3	4
V	3	3	2	1	2	3
E	4	4	3	2	2	2

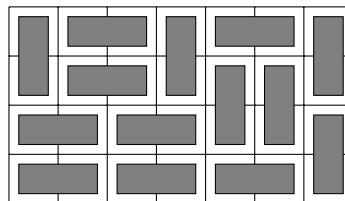
Tablonun sağ alt köşesi bize LOVE ile MOVIE kelimeleri arasındaki değişiklik mesafesinin 2 olduğunu verir. Tablo aynı zamanda en az operasyon sayısını nasıl oluşturacağımızı da gösterir. Bu durumda yol aşağıdaki gibi olur:

		M	O	V	I	E
	0	1	2	3	4	5
L	1	1	2	3	4	5
O	2	2	1	2	3	4
V	3	3	2	1	2	3
E	4	4	3	2	2	2

LOVE ve MOVIE kelimelerinin son karakterleri aynıdır yani aralarındaki değişiklik mesafesi, LOV ile MOVI kelimelerinin değişiklik mesafesine eşittir. I karakterini MOVI kelimesinden bir hamleyle çıkarabiliriz. Böylece değişiklik farkı LOV ve MOV kelimelerinin değişiklik farkından bir fazladır.

7.6 Fayans Sayma (Counting Tilings)

Bazen dinamik programlama çözümünün durumları sabit sayı kombinasyonlarından daha karışık olabilir. Örneğin, amacımızın $n \times m$ 'lik bir düzlemi 1×2 ve 2×1 büyüklüğündeki fayansları kullanarak kaç farklı şekilde doldurabileceğimizi bulmak olsun. Örneğin 4×7 'lik düzlemde geçerli bir çözüm



ve toplam çözüm sayısı 781 olur.

Problem dinamik programlama kullanarak satır satır ilerlemek yöntemiyle çözülebilir. Çözümdeki her satır $\{\sqcap, \sqcup, \sqsubset, \sqsupset\}$ karakterlerinden oluşan bir m karakterli yazı ile gösterilebilir. Örneğin yukarıdaki çözüm 4 satıra karşılık gelir ve aşağıdaki satırla denk gelir:

- $\sqcap \sqsubset \sqsubset \sqcap \sqsubset \sqsubset \sqcap$
- $\sqcup \sqsubset \sqsubset \sqcup \sqcap \sqcap \sqcup$
- $\sqsubset \sqsubset \sqsubset \sqsubset \sqcup \sqcup \sqcap$
- $\sqsubset \sqsubset \sqsubset \sqsubset \sqsubset \sqsubset \sqcup$

$\text{count}(k, x)$, düzlemin $1 \dots k$ 'lık satırlarını kullanarak oluşturulabilecek çözüm sayısını versin ve x yazısı k satırına karşılık gelsin. Burada dinamik programlama kullanmak mümkündür çünkü satırın durumu önceki satırdan gelen durumdan dolayı sınırlandırılmıştır.

Eğer 1. satır \sqcup karakteri ve n . satır \sqcap içermiyorsa ve bütün ardışık satırlar *uyumluysa* çözüm geçerlidir. Örneğin $\sqcup \sqsubset \sqsubset \sqcup \sqcap \sqcap \sqcup$ ve $\sqsubset \sqsubset \sqsubset \sqsubset \sqcup \sqcup \sqcap$ satırları geçerliken $\sqcap \sqsubset \sqcap \sqsubset \sqsubset \sqcap$ ve $\sqsubset \sqsubset \sqsubset \sqsubset \sqsubset \sqsubset \sqcup$ satırları geçerli değildir.

Biz satır m karakterden oluşup her karakter için 4 seçenek olduğu için toplam farklı satır sayısı en fazla 4^m olur. Böylece çözümün toplam zaman karmaşıklığı $O(n4^{2m})$ olur çünkü her satırda $O(4^m)$ tane olası durumdan geçeriz ve her durumda önceki satırdan $O(4^m)$ olası durum vardır. Pratikte düzlemi kısa kenar m olacak şekilde döndürmek mantıklıdır çünkü 4^{2m} faktörü zaman karmaşıklığını domine eder.

Satırları daha sıkışık şekilde göstererek çözümü daha verimli yapmak mümkündür. Aslında önceki satırın hangi sütunlarının bir dikey fayansın üst kısmını içerdiğini bilmek yeterlidir. Böylece satırı sadece \sqcap ve \sqcup karakterleri ile gösterebiliriz. \sqsubset , \sqcup , \sqsubset ve \sqsupset karakterlerinin birer kombinasyonudur. Bu gösterimi kullanarak sadece 2^m tane farklı satır vardır ve zaman karmaşıklığı $O(n2^{2m})$ olur.

Son bir not olarak fayans sayısını hesaplamamanın daha ilginç direkt bir formülü vardır²:

$$\prod_{a=1}^{\lceil n/2 \rceil} \prod_{b=1}^{\lceil m/2 \rceil} 4 \cdot \left(\cos^2 \frac{\pi a}{n+1} + \cos^2 \frac{\pi b}{m+1} \right)$$

Bu formül çok verimlidir çünkü toplam fayans sayısını $O(nm)$ zamanda hesaplar fakat cevap, reel sayıların çarpımı olduğu için formülün sorunu ara değerleri kesin (isabetli) bir şekilde nasıl depolamak olduğudur.

²İlginç bir şekilde bu formül, 1961 yılında birbirlerinden bağımsız çalışan iki araştırma grubu [43, 67] tarafından bulunmuştur.

Bölüm 8

Amortize Analizi

Bir algoritmanın sadece yapısını inceleyerek zaman karmaşıklığını hesaplamak kolaydır: Örneğin, algoritma hangi döngülere sahip ve bu döngüler kaç defa çalışıyor. Fakat, üstünkörü bir analiz algoritmalarının verimliliğini tam doğru yansıtmaz.

Amortize Analizi zaman karmaşıklığı farklı olan operasyonları içeren algoritmaları içerir. Buradaki fikir, tek tek operasyonlara bakmak yerine algoritmanın çalışması sürecinde bütün operasyonlar için harcanan zamanı tahmin etmektir.

8.1 İki İşaretçi Methodu (Two Pointers Method)

İki işaretçi methodunda, dizinin elemanlarından geçmek için iki işaretçi kullanılır. İki işaretçi de tek bir yöne gidebilir ki bu da algoritmanın verimli çalışmasını sağlar. Şimdi iki işaretçi methodu ile çözülebilen sorulara bakacağız.

Aldızı (Subarray) Toplamı

İlk amacımız, n tane pozitif sayıdan oluşan bir dizide x toplamına sahip bir altdizi bulmaktır ve eğer yoksa da belirtmektir.

Örneğin dizimiz

1	3	2	5	1	1	2	3
---	---	---	---	---	---	---	---

toplamı 8 olan bir altdizi içerir:

1	3	2	5	1	1	2	3
---	---	---	---	---	---	---	---

Bu problem iki işaretçi methodu ile $O(n)$ zamanda çözülebilir. Buradaki fikir altdizinin ilk ve son elemanını gösteren işaretçiler tutmaktır. Her adımda, altdizi toplamı en fazla x olduğu sürece sol işaretçiyi bir adım sağa ve sağ işaretçiye de bir adım sağa kaydırmaktır. Eğer toplam tam x oluyorsa bir çözüm bulunmuştur.

Örneğin, aşağıdaki diziyi düşünelim ve hedef toplamımızın $x = 8$ olduğunu düşünelim:

1	3	2	5	1	1	2	3
---	---	---	---	---	---	---	---

İlk altdizi 1, 3 ve 2 değerlerini barındırır ve bunların toplamı 6'dır:

1	3	2	5	1	1	2	3
↑		↑					

Sonra sol işaretçi bir adım sağa kayar. Sağ işaretçi hareket etmez çünkü hareket ederse alt dizi toplamı x 'i geçer.

1	3	2	5	1	1	2	3
	↑	↑					

Yine, sol işaretçi bir adım sağa kayar ve bu sefer sağ işaretçi 3 adım sağa kayar. Altdizi toplamı $2 + 5 + 1 = 8$ olur yani toplamı x olan bir altdizi bulunmuştur.

1	3	2	5	1	1	2	3
		↑	↑				

Algoritmanın çalışma zamanı sağ işaretçinin toplam adım sayısına göre değişir. İşaretçinin *bir* turda atabileceği adım sayısı hakkında sınır yokken, adımın en fazla algoritma sırasında *toplam* $O(n)$ adım atabileceğini biliriz çünkü işaretçi sadece sağa kayar.

Hem sol hem de sağ işaretçiler algoritma sırasında $O(n)$ adım hareket ettikleri için algoritma $O(n)$ zamanda çalışır.

2SUM Problemi

İki işaretçi methodu ile çözülen başka bir soru aşağıdaki gibidir: **2SUM problemi** olarak da bilinen bu problemde n sayıdan oluşan bir dizi ve x toplamı vardır. Amacımız toplamı x olan iki değer bulmaktır ve eğer yoksa da belirtmektir.

Problemi çözmek için diziyi artan sırada sıralarız. Sonra diziyi iki işaretçi yöntemi ile gezeriz. Sol işaretçi ilk elemanda başlar her adımda bir sağa gider. Sağ işaretçi ise en son elemanda başlar ve sol ile sağdaki değerlerin toplamı en fazla x olana kadar sola gider. Eğer tam x toplamı bulunduyorsa bir çözüm bulunmuştur.

Örneğin, aşağıdaki diziyi ve hedef toplamımızın $x = 12$ olduğunu düşünelim:

1	4	5	6	7	9	9	10
---	---	---	---	---	---	---	----

İşaretçilerin pozisyonları şekildeki gibidir. Değerlerin toplamı $1 + 10 = 11$ olur ve bu da x 'ten küçüktür.

1	4	5	6	7	9	9	10
↑							↑

Sonra sol işaretçi bir sağa kayar. Sağ işaretçi bir sola kayar ve toplam $4+7 = 11$ olur.

1	4	5	6	7	9	9	10
	↑			↑			

Bundan sonra, sol işaretçi yeniden bir adım sağa gider. Sağ işaretçi hareket etmez ve $5+7 = 12$ çözümü bulunur.

1	4	5	6	7	9	9	10
		↑		↑			

Algoritmanın çalışma zamanı $O(n \log n)$ olur çünkü algoritma ilk diziyi $O(n \log n)$ zamanda sıralar ve her iki işaretçi $O(n)$ adım atar.

Bu soruyu ikili arama ile $O(n \log n)$ zamanda çözmek mümkündür. Bu yöntemde diziyi dolaşırız ve her dizi değeri için toplamı x veren başka bir değer bulmaya çalışırız. Bu n ikilli arama ile olur ve her bir tanesi $O(\log n)$ zaman harcar.

Daha zor bir problem ise **3SUM problemi** yani toplamı x olan üç tane dizi elemanı bulmaktır. Yukarıdaki algoritmadaki fikri kullanarak bu problem $O(n^2)$ zamanda çözülebilir¹. Nasıl yapılabileceğini görüyor musunuz?

8.2 En Yakın Küçük Elemanlar (Nearest Smaller Elements)

Bir veri yapısında yapılan operasyon sayısını tahmin etmek için genelde amortize analizi kullanılır. Operasyonlar bazen çok farklı bir şekilde dağıtılabilir ki mesela algoritmanın bir kısmında çok fazla operasyon olabilir fakat toplam operasyon sayısı sınırlıdır.

Örneğin her elemanın **en yakın küçük elemanını** bulmamız gerektiğini yani elemanın sol tarafında bulunup kendisinden küçük olan en yakın elemanı bulmamız gerekir. Bu tip bir eleman olmayabilir ki bu durumda algoritma bunu belirtmesi gerekir. Şimdi bu problemi verimli bir şekilde veri yapısı kullanarak çözeceğiz.

Dizinin solundan sağına doğru geçeriz ve dizi elemanlarından oluşan yığın(stack) oluştururuz. Her dizi pozisyonunda, yığındaki en üstte bulunan eleman şu anki elemandan küçük olana kadar ya da öyle bir sayı yoksa yığın boşalana kadar yığından eleman çıkarırız. Sonra en üstteki elemanın şu anki elemanın en yakın küçük elemanı olduğunu söyleriz ve eğer yığın boşsa öyle bir elemanın bulunmadığını söyleriz. En sonda, şu anki diziyi yığına ekleriz.

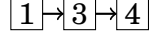
Örneğin aşağıdaki diziyi düşünelim:

¹Uzun bir zaman boyunca 3SUM problemini $O(n^2)$ zamandan daha verimli bir şekilde çözenin mümkün olmayacağı kabul edilmiştir. Fakat 2014'te [30] bu durumun böyle olmadığı anlaşıldı

1	3	4	2	5	3	4	2
---	---	---	---	---	---	---	---

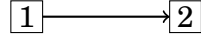
İlk 1, 3 ve 4 elemanları yığına eklenir çünkü her eleman önceki elemandan daha büyüktür. Böylece 4'ün en yakın küçük elemanı 3, ve 3'ün en yakın küçük elemanı 1 olur.

1	3	4	2	5	3	4	2
---	---	---	---	---	---	---	---



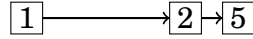
Sonraki eleman 2, yığının en üstteki iki elemanından küçüktür. Bu yüzden 3 ve 4 yığından çıkartılır ve sonra 2 elemanı yığına eklenir. Onun en yakın küçük elemanı 1'dir:

1	3	4	2	5	3	4	2
---	---	---	---	---	---	---	---



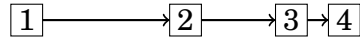
Sonra 5 elemanı 2 elemanından daha büyüktür ki bu da yığına eklenir ve en yakın küçük elemanı 2 olur:

1	3	4	2	5	3	4	2
---	---	---	---	---	---	---	---



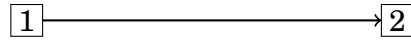
Bundan sonra 5 elemanı yığından çıkarılır ve 3 ile 4 elemanları yığına eklenir:

1	3	4	2	5	3	4	2
---	---	---	---	---	---	---	---



En sonda 1 hariç bütün elemanlar yığından çıkarılır ve en sonki eleman yani 2 yığına eklenir:

1	3	4	2	5	3	4	2
---	---	---	---	---	---	---	---



Algoritmanın verimliliği toplam yığın operasyonu sayısına bağlıdır. Eğer şu anki eleman yığının en üstündeki elemandan büyükse, direk yığına eklenir ki bu verimlidir. Fakat bazen yığın birkaç daha büyük eleman içerir ve onları çıkarmak zaman alır. Yine de her elemene *tam bir defa* yığına eklenir ve *tam bir defa* yığından çıkarılır. Böylece her eleman $O(1)$ yığın operasyonu içerir ve algoritma $O(n)$ zamanda çalışır.

8.3 Sürgülü Pencere Minimumu (Sliding Window Minimum)

Bir **sürgülü pencere** sabit boyutta alt dizi olup dizide soldan sağa doğru ilerler. Her pencere pozisyonunda pencere içindeki elemanlar hakkında bilgi almak isteriz. Bu bölümde **sürgülü pencere minimumu** tutmaya odaklanacağız yani her pencere içindeki en küçük sayıyı söyleyeceğiz.

Sürgülü pencere minimumu en yakın küçük elemanı hesapladığımıza benzer bir şekilde hesaplayabiliriz. Her elemanın önceki elemandan büyük olduğu bir kuyruk yapısı tutarız ve ilk eleman her zaman pencere içindeki minimum elemana karşılık gelir. Her pencere hareketinden sonra, en son kuyruk elemanı yeni pencere elemanından küçük olana kadar ya da öyle bir eleman yoksa kuyruk boşalana kadar kuyruğun sonundan eleman çıkarırız. Aynı zamanda artık pencerede bulunmuyorsa ilk kuyruk elemanını çıkarırız. En son yeni pencere elemanını kuyruğa ekleriz.

Örneğin, aşağıdaki diziyi düşünelim:

2	1	4	5	3	4	1	2
---	---	---	---	---	---	---	---

Diyelim ki sürgülü pencerenin boyu 4 olsun. İlk pencere pozisyonunda en küçük değer 1'dir:

2	1	4	5	3	4	1	2
---	---	---	---	---	---	---	---

1	→	4	→	5
---	---	---	---	---

Sonra, pencere bir adım sağa kayar. Yeni eleman olan 3, kuyruktaki 4 ve 5 elemanlarından daha küçüktür. Bu yüzden 4 ve 5 elemanları kuyruktan çıkarılır ve 3 elemanı kuyruğa eklenir. En küçük değer hala 1'dir.

2	1	4	5	3	4	1	2
---	---	---	---	---	---	---	---

1	→	3
---	---	---

Bundan sonra pencere yeniden hareket eder ve en küçük eleman olan 1 herhangi bir pencereye ait olmaz. Böylece kuyruktan çıkarılır ve en küçük eleman artık 3 olur. Aynı zamanda 4 kuyruğa eklenir.

2	1	4	5	3	4	1	2
---	---	---	---	---	---	---	---

3	→	4
---	---	---

Sonraki yeni eleman 1, kuyruktaki bütün elemanlardan küçüktür. Böylece kuyruktaki bütün elemanlar çıkarılır ve sadece 1 elemanını içerir:

2	1	4	5	3	4	1	2
---	---	---	---	---	---	---	---

1

En sonda pencere son pozisyonuna ulaşır. 2 elemanı kuyruğa eklenir ama penceredeki en küçük değer hala 1'dir.

2	1	4	5	3	4	1	2
---	---	---	---	---	---	---	---

1 → 2

Her dizi elemanı kuyruğa tam bir defa eklenip en fazla bir defa çıkarıldığından algoritma $O(n)$ zamanda çalışır.

Bölüm 9

Aralık Sorguları

Bu bölümde aralık sorguları verimli bir şekilde yapmamızı sağlayan veri yapılarından bahsedeceğiz. Bir **aralık sorgusunda** görevimiz bir dizinin alt dizisinde bir değeri hesaplamaktır. Tipik aralık sorguları:

- $\text{sum}_q(a, b)$: $[a, b]$ aralığındaki sayıların toplamını bul
- $\text{min}_q(a, b)$: $[a, b]$ aralığındaki minimum sayıyı bul
- $\text{max}_q(a, b)$: $[a, b]$ aralığındaki maksimum sayıyı bul

Örneğin aşağıdaki dizide $[3, 6]$ aralığını düşünün:

0	1	2	3	4	5	6	7
1	3	8	4	6	1	3	4

Bu durumda $\text{sum}_q(3, 6) = 14$, $\text{min}_q(3, 6) = 1$ ve $\text{max}_q(3, 6) = 6$.

Aralık sorgularını işlemenin basit bir yolu istenen aralığın içindeki bütün dizi değerlerinden geçen bir döngü yapmaktır. Örneğin aşağıdaki fonksiyon, bir dizideki toplam sorgularını işlemek için kullanılabilir:

```
int sum(int a, int b) {  
    int s = 0;  
    for (int i = a; i <= b; i++) {  
        s += array[i];  
    }  
    return s;  
}
```

Bu fonksiyon n 'in dizi büyüklüğü olduğu durumda $O(n)$ zamanda çalışır. Böylece q tane sorguyu $O(nq)$ zamanda fonksiyonu kullanarak işleyebiliriz. Fakat n ve q sayılarının ikisi de büyükse bu çözüm yavaş kalır. Neyseki aralık sorgularını daha verimli işlemenin yolları var.

9.1 Statik Dizi Sorguları

İlk başta dizinin *statik* (dizi değerlerinin sorgularda güncellenmediği) olduğu bir duruma bakacağız. Bu durumda bize herhangi bir sorgu için cevabı veren sabit bir veri yapısı oluşturmak yeterlidir.

Toplam Sorguları

Statik bir dizideki toplam sorgularını rahat bir şekilde **prefix toplam dizisi**yle oluşturabiliriz. Prefix toplamı dizisindeki her değer orijinal dizide o pozisyona kadar olan değerlerin toplamına eşittir yani k pozisyonundaki değer $\text{sum}_q(0, k)$. Prefix toplam dizisi $O(n)$ zamanda oluşturulabilir.

Örneğin aşağıdaki diziyi düşünelim:

0	1	2	3	4	5	6	7
1	3	4	8	6	1	4	2

Buna karşılık gelen prefix toplam dizisi aşağıdaki gibidir:

0	1	2	3	4	5	6	7
1	4	8	16	22	23	27	29

Prefix toplamı dizisi bütün $\text{sum}_q(0, k)$ değerlerini içerdiği için herhangi bir $\text{sum}_q(a, b)$ değerini $O(1)$ zamanda aşağıdaki gibi bulabiliriz:

$$\text{sum}_q(a, b) = \text{sum}_q(0, b) - \text{sum}_q(0, a - 1)$$

$\text{sum}_q(0, -1) = 0$ tanımlayarak yukarıdaki formü $a = 0$ için de olur.

Örneğin $[3, 6]$ aralığını düşünelim:

0	1	2	3	4	5	6	7
1	3	4	8	6	1	4	2

Bu durumda $\text{sum}_q(3, 6) = 8 + 6 + 1 + 4 = 19$ olur. Bu toplam prefix toplam dizisinin iki değeri kullanılarak bulunabilir:

0	1	2	3	4	5	6	7
1	4	8	16	22	23	27	29

Böylece $\text{sum}_q(3, 6) = \text{sum}_q(0, 6) - \text{sum}_q(0, 2) = 27 - 8 = 19$ olur.

Bu fikri daha fazla boyutlara da genelleştirebiliriz. Örneğin iki boyutlu bir prefix toplam dizisi oluşturularak herhangi bir dikdörtgen alt dizi toplamını $O(1)$ zamanda hesaplayabiliriz. Bu tip bir dizideki her toplam, dizinin sol üst köşesinden başlayan bir alt diziye karşılık gelir.

Aşağıdaki resim bu fikri gösteriyor:

		<i>D</i>				<i>C</i>			
		<i>B</i>				<i>A</i>			

Gri alt dizinin toplamı, $S(X)$ 'in sol üst köşeden X pozisyonuna gelen dikdörtgen alt dizinin değerlerinin toplamına denk geldiği varsayılarak

$$S(A) - S(B) - S(C) + S(D)$$

formülü ile hesaplanabilir.

Minimum Sorgular

Minimum sorgular toplam sorgularını işlemeye göre daha zordur. Yine de kolay $O(n \log n)$ 'lik bir ön işleme methodu ile minimum sorguları $O(1)$ zamanda hesaplayabiliriz¹. Bu arada minimum ve maksimum sorgular benzer bir şekilde işlenebildiğinden minimum sorgulara bakacağız.

Buradaki fikir $b - a + 1$ (aralığın uzunluğu) değerinin ikinin kuvveti olan bütün $\min_q(a, b)$ değerlerini ön işleme ile işleriz. Örneğin

0	1	2	3	4	5	6	7
1	3	4	8	6	1	4	2

dizisinde aşağıdaki değerler hesaplanır:

a	b	$\min_q(a, b)$	a	b	$\min_q(a, b)$	a	b	$\min_q(a, b)$
0	0	1	0	1	1	0	3	1
1	1	3	1	2	3	1	4	3
2	2	4	2	3	4	2	5	1
3	3	8	3	4	6	3	6	1
4	4	6	4	5	1	4	7	1
5	5	1	5	6	1	0	7	1
6	6	4	6	7	2			
7	7	2						

Ön işleme ile işlenen değerlerin sayısı $O(n \log n)$ olur çünkü ikinin kuvveti olan $O(\log n)$ tane aralık uzunluğu vardır. Bu değerler verimli bir şekilde

$$\min_q(a, b) = \min(\min_q(a, a + w - 1), \min_q(a + w, b))$$

¹Bu teknik [7] kaynağında tanıtılmıştır ve bazen **sparse table** mehodu olarak söylenir. Bunla beraber daha gelişmiş teknikler de [22] vardır ki burada ön işleme sadece $O(n)$ zaman alır fakat bu tip algoritmalar rekabetçi programlamada gerekli değildir.

özyinelemeli formülü ile hesaplanabilir ve burada $b-a+1$ ikinin kuvveti olup $w = (b-a+1)/2$ olur. Bütün bu değerleri hesaplamak $O(n \log n)$ zaman alır. Bundan sonra herhangi bir $\min_q(a, b)$ değerini $O(1)$ zamanda iki tane önceden işlenmiş değerlerin minimumu olarak bulabiliriz. Diyeim ki k , $b-a+1$ değerini geçmeyen en büyük ikinin kuvvet olsun. $\min_q(a, b)$ değerini

$$\min_q(a, b) = \min(\min_q(a, a+k-1), \min_q(b-k+1, b))$$

formülü ile hesaplayabiliriz. Yukarıdaki formülde $[a, b]$ aralığı, ikisi de k uzunluğunda olan $[a, a+k-1]$ ve $[b-k+1, b]$ aralıklarının birleşimi ile gösterilebilir.

Örneğin $[1, 6]$ aralığını düşünelim:

0	1	2	3	4	5	6	7
1	3	4	8	6	1	4	2

Aralığın uzunluğu 6'dır ve 6'yı geçmeyen en büyük ikinin kuvveti 4'tür. Böylece $[1, 6]$ aralığı $[1, 4]$ ile $[3, 6]$ aralıklarının birleşimi olarak gösterilebilir:

0	1	2	3	4	5	6	7
1	3	4	8	6	1	4	2

0	1	2	3	4	5	6	7
1	3	4	8	6	1	4	2

$\min_q(1, 4) = 3$ ve $\min_q(3, 6) = 1$ olduğu için $\min_q(1, 6) = 1$ diyebiliriz.

9.2 İkili İndisli Ağaç (Binary Indexed Tree)

Bir **ikili indisli ağaç** veya **Fenwick ağacı**², prefix toplam dizisinin dinamik tipi olarak görülebilir. Bir dizi üzerinde $O(\log n)$ zamanda çalışan iki operasyonu destekler: aralık toplamını bulmak veya bir değeri güncellemek.

İkili indisli ağacın yararı bize toplam sorguları arasında dizi değerlerini güncelleyebilmemizi sağlamasıdır. Bu prefix toplam dizisinde mümkün olmaz çünkü her güncelleden sonra bütün prefix toplam dizisini $O(n)$ zamanda en baştan oluşturmamız gerekir.

Yapı

Her ne kadar apının ismi ikili indisli *ağaç* olsa da genelde dizi şeklinde gösterilir. Bu bölümde bütün dizilerin indislerinin 1'den başladığını varsayacağız çünkü bu implementasyonu kolaylaştırıyor.

$p(k)$ k 'yi bölebilen en büyük 2'nin kuvveti olsun. Bütün ikili indis ağacını *tree* dizisinde öyle bir tutarız ki

$$\text{tree}[k] = \text{sum}_q(k - p(k) + 1, k)$$

²İkili indisli ağaç yapısı P. M. Fenwick tarafından 1994'te tanıtılmıştır [21].

yani her k pozisyonu, uzunluđu $p(k)$ olup k pozisyonunda biten aralıkların toplamını bulundurur. Örneğin $p(6) = 2$ olduđu için $\text{tree}[6]$ ifadesi $\text{sum}_q(5,6)$ değerini barındırır.

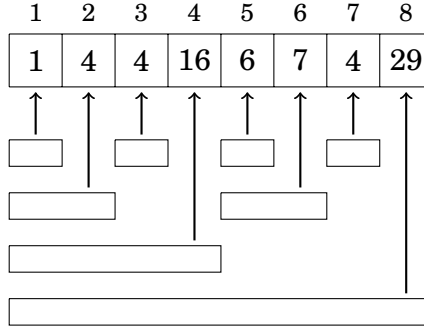
Örneğin aşağıdaki diziye

1	2	3	4	5	6	7	8
1	3	4	8	6	1	4	2

karşılık gelen ikili indisli ağaç şeklindeki gibidir:

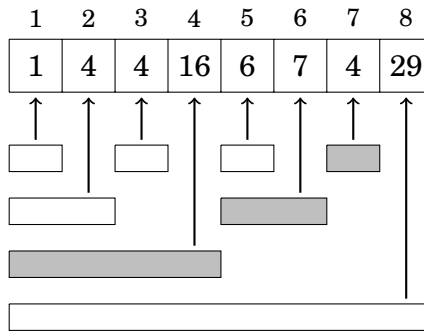
1	2	3	4	5	6	7	8
1	4	4	16	6	7	4	29

Aşağıdaki resim, ikili indisli ağaçtaki her değerin orijinal dizide bir aralığa geldiğini daha güzel şekilde gösteriyor:



İkili indisli ağaç kullanarak herhangi bir $\text{sum}_q(1,k)$ değeri $O(\log n)$ zamanda hesaplanabilir çünkü $[1,k]$ aralığı her zaman ağaçta toplamaları tutulan $O(\log n)$ aralığa bölünebilir.

Örneğin $[1,7]$ aralığı aşağıdaki aralıklardan oluşur:



Böylece karşılık gelen toplamı aşağıdaki gibi hesaplarız:

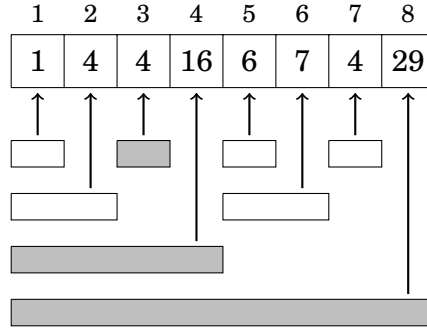
$$\text{sum}_q(1,7) = \text{sum}_q(1,4) + \text{sum}_q(5,6) + \text{sum}_q(7,7) = 16 + 6 + 4 = 27$$

$a > 1$ olduđu durumda $\text{sum}_q(a,b)$ değerini hesaplamak için prefix toplam dizilerinde kullandığımız taktiği kullanırız:

$$\text{sum}_q(a,b) = \text{sum}_q(1,b) - \text{sum}_q(1,a-1).$$

Hem $\text{sum}_q(1, b)$ hem de $\text{sum}_q(1, a - 1)$ değerlerini $O(\log n)$ zamanda hesaplayabildiğimiz için toplam zaman karmaşıklığı $O(\log n)$ olur.

Bundan sonra orijinal dizide bir değeri güncellerken ikili indisli ağaçtaki bazı değerler güncellenir. Örneğin eğer 3. pozisyonadaki değer değişirse, aşağıdaki aralıkların toplamı değişir:



Her dizi elemanı ikili indisli ağaçta $O(\log n)$ tane aralığa karşılık geldiği için ağaçta $O(\log n)$ tane değeri güncellemek yeterli oluyor.

Implementasyon

İkili indisli ağaçtaki operasyonlar verimli bir şekilde bit operasyonları ile koda geçirilebilir. Herhangi bir $p(k)$ değerini

$$p(k) = k \& -k.$$

ile hesaplayabiliriz. Aşağıdaki fonksiyon $\text{sum}_q(1, k)$ değerini hesaplar:

```
int sum(int k) {
    int s = 0;
    while (k >= 1) {
        s += tree[k];
        k -= k & -k;
    }
    return s;
}
```

Aşağıdaki fonksiyon k . pozisyonadaki dizinin elemanını x kadar arttırır (x pozitif veya negatif olabilir.):

```
void add(int k, int x) {
    while (k <= n) {
        tree[k] += x;
        k += k & -k;
    }
}
```

İki fonksiyonun da zaman karmaşıklığı $O(\log n)$ olur çünkü fonksiyon ikili indisli ağaçta $O(\log n)$ tane değere erişir ve sonraki pozisyona olan her hareket $O(1)$ zaman alır.

9.3 Segment Ağacı

Segment ağacı³, iki operasyonu destekleyen bir veri yapısıdır: bir aralık sorgusunu işlemek ve bir dizi değerini güncellemek. Segment ağacı aralık toplamı sorgularını, minimum ve maksimum sorgularını ve $O(\log n)$ zamanda çalışan bir sürü operasyonu destekler.

İkili indisli ağaçla karşılaştırıldığında zaman segment ağacının avantajı daha genel bir veri yapısı olmasıdır. İkili indisli ağaç sadece toplam sorgularını desteklerken⁴ segment ağacı başka sorguları da destekler. Diğer taraftan bir segment ağacı daha fazla hafıza gerektirir ve koda geçirilmesi bir tık daha zordur.

Yapı

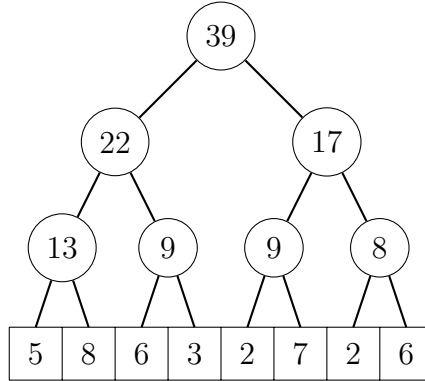
Segment ağacı bir ikili ağaç olup ağacın en alt seviyesindeki düğümler dizi elemanlarına karşılık gelir ve diğer elemanlar aralık sorgularını uygulamak için gerekli olan bilgileri içerir.

Bu bölümde, dizinin ikinin kuvveti büyüklüğünde olduğunu ve sıfır tabanlı indis kullanıldığı varsayılacaktır çünkü böyle bir diziye segment ağacı oluşturmak daha kolaydır. Eğer dizi ikinin kuvveti değilse her zaman fazla elemanları ekleyebiliriz.

İlk başta toplam sorgularını destekleyen segment ağaçlarına bakacağız. Örneğin aşağıdaki diziye:

0	1	2	3	4	5	6	7
5	8	6	3	2	7	2	6

karşılık gelen segment ağacı aşağıdaki gibidir:



Her iç ağaç düğümü büyüklüğü ikinin kuvveti olan bir dizi aralığına karşılık gelmektedir. Yukarıdaki ağaçta her iç düğümün değeri ona karşılık gelen dizinin elemanlarının toplamıdır ve bu değer soldaki ve sağdaki çocuklarının değerlerinin toplamı olarak bulunabilir.

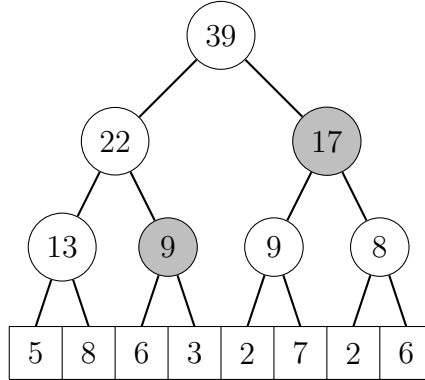
³Bu bölümdeki aşağıdan yukarıya implementasyon [62] kaynağına karşılık gelir. 1970'lerin sonlarında geometri problemleri çözmek için benzer yapılar kullanılmıştır [9].

⁴Aslında *iki* tane ikili indisli ağaç kullanarak minimum sorgularını yaptırmak mümkündür [16] fakat bu segment ağacı kullanmaya göre daha karışıktır.

Herhangi bir $[a, b]$ aralığı değerleri ağaç düğümlerinde bulunan $O(\log n)$ tane aralığa bölünebilir. Örneğin $[2, 7]$ aralığında:

0	1	2	3	4	5	6	7
5	8	6	3	2	7	2	6

Burada $\text{sum}_q(2, 7) = 6 + 3 + 2 + 7 + 2 + 6 = 26$. Bu durumda aşağıdaki iki ağaç düğümü aşağıdaki aralığa karşılık gelir:

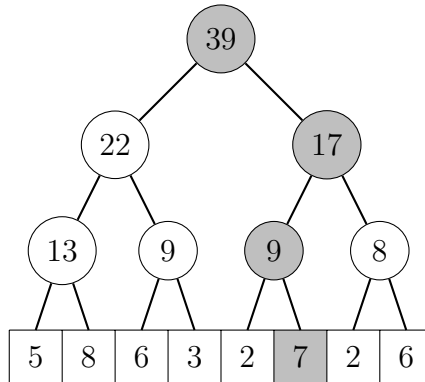


Böylece toplamaları hesaplamamızın başka bir yolu $9 + 17 = 26$.

Ağaçtaki olabilecek en yüksekteki düğümleri kullanarak toplamı hesaplarken, ağaçtaki her seviyeden en fazla iki düğüm gereklidir. Bu yüzden toplam düğüm sayısı $O(\log n)$ olur.

Bir dizi güncellemesinden sonra güncellenmiş değere bağlı olan bütün değerleri güncellememiz gerekir. Bu da güncellenmiş dizi elemanından en yüksek düğüme giden yolu takip yoldaki düğümleri güncelleyerek mümkündür.

Aşağıdaki resim 7 dizinin değeri değişince hangi ağaç düğümlerinin değiştiğini gösteriyor:



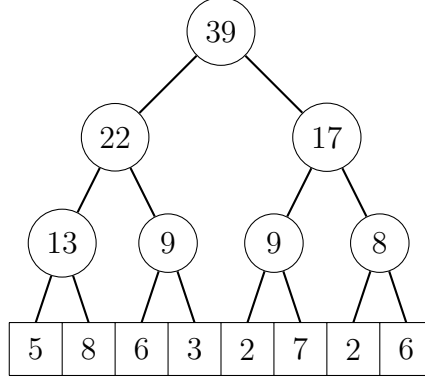
Aşağıdan yukarıya olan yol her zaman $O(\log n)$ düğüm içerir yani ağaçta her güncelleme $O(\log n)$ düğüm değiştirir.

Implementasyon

Segment ağacını, orijinal dizinin boyutu n ve ikinin kuvveti olduğu durumda $2n$ tane elemanda oluşan bir dizide tutarız. Ağaç düğümleri yukarıdan aşağıya

tutulur: $tree[1]$ tepe düğüme, $tree[2]$ ve $tree[3]$ tepe düğümün çocukları gibi gider. En sonda $tree[n]$ 'den $tree[2n - 1]$ 'e kadar olan değerler ağacın en alt seviyesindeki düğümler olup orijinal dizideki değerlere karşılık gelirler.

Örneğin



segment ağacı şeklindeki gibi tutulur:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
39	22	17	13	9	9	8	5	8	6	3	2	7	2	6

Bu gösterimi kullanarak $tree[k]$ 'nin ebebeyn düğümü $tree[\lfloor k/2 \rfloor]$ ve çocukları $tree[2k]$ ve $tree[2k+1]$ olur. Bu durum, düğümün pozisyonunun eğer sol çocuksa çift sayı olduğunu ve eğer sağ çocuksa tek sayı olduğunu gösterir.

Aşağıdaki fonksiyon $sum_q(a, b)$ değerini hesaplar:

```
int sum(int a, int b) {
    a += n; b += n;
    int s = 0;
    while (a <= b) {
        if (a%2 == 1) s += tree[a++];
        if (b%2 == 0) s += tree[b--];
        a /= 2; b /= 2;
    }
    return s;
}
```

Bu fonksiyon başta $[a+n, b+n]$ olan bir aralık tutar. Sonra her adımda aralık ağaçtaki bir üst seviyeye çıkarılır ve ondan önce üstteki aralığa ait olmayan değerler toplama eklenir.

Aşağıdaki fonksiyon k pozisyonundaki elemanın değerini x arttırır:

```
void add(int k, int x) {
    k += n;
    tree[k] += x;
    for (k /= 2; k >= 1; k /= 2) {
        tree[k] = tree[2*k] + tree[2*k+1];
    }
}
```

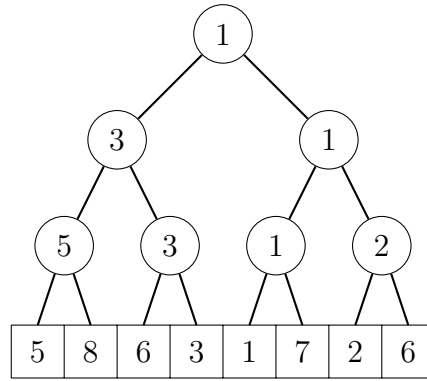
İlk başta fonksiyon ağacının en alt seviyesindeki değeri günceller. Sonra fonksiyon ağacının tepe düğümüne kadar geldiği bütün iç ağaç düğümlerindeki değerleri günceller.

Yukarıdaki fonksiyonlar $O(\log n)$ zamanda çalışır çünkü n elemanlık bir segment ağacı $O(\log n)$ tane seviye içerir ve fonksiyon her adımda ağacın bir üst seviyesine çıkar.

Diğer Sorgular

Segment ağaçları aralığı iki parçaya bölmenin mümkün olduğu bütün aralık sorgularını destekler. Aralığı bölerek cevapları ayrı ayrı parçalarda bulur ve en sonda verimli bir şekilde cevapları birleştirir. Bu tip sorgulardan bazıları minimum ve maksimum, en büyük ortan bölen ve bit operasyonları olan and (ve), or (veya), xor (ya da) olur.

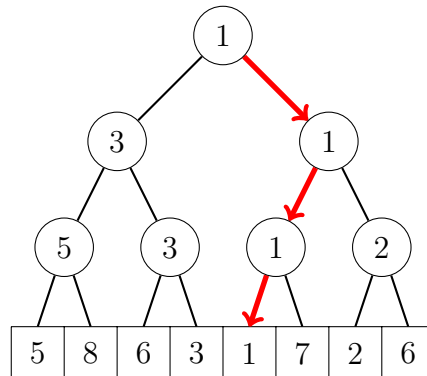
Örneğin aşağıdaki dizi minimum sorgularını destekler:



Bu durumda her ağaç düğümü ona karşılık gelen aralıktaki en küçük değeri tutar. Ağacın tepe noktası da bu yüzden bütün dizideki en küçük elemanı tutar. Bu operasyonlar önceki implemantasyon gibi koda geçirilebilir fakat bu sefer toplamları hesaplamak yerine minimum bulunur.

Segment ağacının yapısı, bize dizi elemanlarını bulurken ikili aramayı (binary search) kullanabilmemizi sağlar. Örneğin ağaç minimum sorgularını destekliyorsa en küçük değere sahip elemanı $O(\log n)$ zamanda bulabiliriz.

Örneğin yukarıdaki ağaçta, en küçük değer olan 1'e sahip bir eleman, yolda tepe düğümüne aşağıya doğru gezilerek bulunabilir:



9.4 Ek Teknikler

Indis Sıkıştırması

Veri yapılarının sınırlarından biri ardışık sayılarla oluşturulmuş diziler üzerine kurulmuş olmasıdır. Büyük tam indisler gerektiği zaman sıkıntılar baş göstermektedir. Örneğin 10^9 indisini kullanmak istediğimiz zaman dizi 10^9 tane eleman içermelidir ki bu da çok fazla hafıza harcar.

Fakat bu sınırı **indis sıkıştırması** kullanarak geçebiliriz. Burada orijinal indisler **1,2,3** gibi indislerle değişiyor. Bunu yapabilmemiz için algoritma sırasında bütün indisleri biliyor olmamız lazımdır.

Buradaki fikir orijinal x indisini $c(x)$ indisile değiştirmektir ki c indisleri bastıran bir fonksiyondur. İndislerin sırasının değişmeyeceğini bilmemiz gerekir yani eğer $a < b$ ise $c(a) < c(b)$ olmalıdır. Bu bize indisler sıkıştırılmış olsa bile sorguları rahatlıkla yapabilmemizi sağlar.

Örneğin orijinal indisler **5555**, 10^9 ve **8** ise yeni indisler:

$$\begin{aligned}c(8) &= 1 \\c(555) &= 2 \\c(10^9) &= 3\end{aligned}$$

Aralık Güncellemeleri

Şu ana kadar aralık sorgularını ve tek eleman güncelleyebilen veri yapılarını oluşturduk. Şimdi de tam tersi durumunu düşünelim yani aralıkları güncelleyip tek eleman almamız gerektiği durumları. Burada $[a, b]$ aralığındaki bütün elemanları x arttıran bir operasyona odaklanacağız.

İlginç bir şekilde, bu durum için bu bölümde gösterilen veri yapılarını kullanabiliriz. Bunu yapmak için bir **fark dizisi** oluştururuz ki buradaki değerler şu anki değerler ile orijinal değer arasındaki farkı verir. Böylece orijinal dizi, fark dizisinin prefix toplam dizisi olur. Örneğin aşağıdaki diziyi düşünelim:

0	1	2	3	4	5	6	7
3	3	1	1	1	5	2	2

Yukarıdaki dizinin fark dizisi aşağıdaki gibi olur:

0	1	2	3	4	5	6	7
3	0	-2	0	0	4	-3	0

Örneğin, orijinal dizideki 6. pozisyonundaki 2 değeri, fark dizisinde $3-2+4-3=2$ toplamına karşılık gelir.

Fark dizisinin avantajı, orijinal dizideki bir aralığı fark dizisindeki sadece iki elemanı kullanarak değiştirebiliyor olmamızdır. Örneğin eğer orijinal dizideki 1 ve 4. pozisyonları arasındaki değerleri 5 arttırmak istiyorsak 1. pozisyonundaki değeri 5 arttırıp 5. pozisyonundaki değeri 5 azaltmamız yeterlidir. Sonuç aşağıdaki gibi olur:

0	1	2	3	4	5	6	7
3	5	-2	0	0	-1	-3	0

Daha genel olarak, $[a, b]$ aralığındaki değerleri x kadar arttırmak için a pozisyonundaki değeri x arttırıp $b + 1$ pozisyonundaki değeri x azaltırız. Böylece tek elemanı güncellerken ve toplam sorguları işlerken ikili indisli ağaç veya segment ağacı kullanırız.

Daha zor bir problem ise hem aralık sorgularını hem de aralık güncellemelelerini destekleyen bir yapı oluşturmaktır. Bölüm 28'de bunun mümkün olduğunu göreceğiz.

Unsigned gösteriminde, sadece negatif olmayan sayılar kullanılır ama değerlerin üst sınırı daha fazla olur. n bitlik bir unsigned değişken 0 ile $2^n - 1$ arası herhangi bir sayı olabilir. Örneğin C++’da bir unsigned `int` değişkeni 0 ve $2^{32} - 1$ arası bir tam sayı olabilir.

Gösterimler arasında bir bağlantı var: Bir signed sayı $-x$, bir unsigned sayıya $2^n - x$ eşittir. Örneğin, aşağıdaki kod $x = -43$ signed sayısı, $y = 2^{32} - 43$ unsigned sayısına eşit olduğunu gösterir:

```
int x = -43;
unsigned int y = x;
cout << x << "\n"; // -43
cout << y << "\n"; // 4294967253
```

Eğer bir sayı, bit gösteriminin üst sınırından daha fazlaysa, sayı overflow verir. Signed gösteriminde, $2^{n-1} - 1$ ’den sonraki sayı -2^{n-1} olur ve unsigned gösterimde $2^n - 1$ ’den sonraki sayı 0 olur. Örneğin aşağıdaki kodda:

```
int x = 2147483647
cout << x << "\n"; // 2147483647
x++;
cout << x << "\n"; // -2147483648
```

İlk başta, x ’in değeri $2^{31} - 1$. Bu `int` değişkenin alabileceği en büyük değerdir yani $2^{31} - 1$ ’den sonraki sayı -2^{31} olur.

10.2 Bit Operasyonu

Ve operasyonu

Ve (and) operasyonu $x \& y$ hem x ’in hem de y ’in 1 olduğu bitlerde 1, diğer durumlarda 0’dan oluşan bir sayı oluşur. Örneğin $22 \& 26 = 18$ çünkü

$$\begin{array}{r} 10110 \quad (22) \\ \& \quad 11010 \quad (26) \\ \hline = \quad 10010 \quad (18) \end{array}$$

Ve gösterimini kullanarak, bir x sayısının çift olup olmadığını kontrol edebiliriz çünkü x çift ise $x \& 1 = 0$ ve x tek sayıysa $x \& 1 = 1$ olur. Genel olarak $x \& (2^k - 1) = 0$ ise x , 2^k sayısına bölünebiliyordur.

Veya operasyonu

Veya operasyonu $x \mid y$, x veya y sayılarının en az birinde 1 biti içerdiği pozisyonlarda 1, diğer bitlerde 0 bit içeren bir sayı oluşturur. Örneğin, $22 \mid 26 = 30$ olur çünkü

$$\begin{array}{r} 10110 \quad (22) \\ \mid \quad 11010 \quad (26) \\ \hline = \quad 11110 \quad (30) \end{array}$$

xor operasyonu $x \wedge y$ ya da y sayılarından sadece bir tanesi 1 biti bulunduğu yerlerde 1, diğer durumlarda 0 içeren bir sayı oluşturur. Örneğin, $22 \wedge 26 = 12$ olur çünkü

103

Fonksiyonlar

g++ derleyicisi bitleri saymak aşağıdaki fonksiyonları barındırır:

- `__builtin_clz(x)`: sayının başında olan 0 sayısını verir.
- `__builtin_ctz(x)`: sayının sonunda olan 0 sayısını verir.
- `__builtin_popcount(x)`: sayıdaki 1 sayısını verir.
- `__builtin_parity(x)`: 1 bit sayısının çift veya tek olduğunu verir.

Fonksiyonlar aşağıdaki gibi kullanılabilir:

```
int x = 5328; // 0000000000000000000000001010011010000
cout << __builtin_clz(x) << "\n"; // 19
cout << __builtin_ctz(x) << "\n"; // 4
cout << __builtin_popcount(x) << "\n"; // 5
cout << __builtin_parity(x) << "\n"; // 1
```

Yukarıdaki fonksiyonlar sadece `int` sayılarını destekliyor olsa da fonksiyonlara 11 ön eki ekleyerek `long long` sayılarına uygun fonksiyonlar da vardır.

10.3 Kümeleri Göstermek

Bir $\{0, 1, 2, \dots, n-1\}$ kümenin her alt kümesi n tane bitten oluşan bir tamsayıda gösterilebilir. 1 bitleri alt kümeye ait olan elemanları belirtir. Bu kümeleri göstermek için verimli bir yoldur çünkü her eleman bir bit hafıza gerektirir ve küme operasyonları bit operasyonları olarak yapılabilir.

Örneğin `int` 32-bit olduğu için bir `int` sayısı, bir $\{0, 1, 2, \dots, 31\}$ kümesinin herhangi bir alt kümesini gösterebilir. $\{1, 3, 4, 8\}$ kümesinin bit gösterimi

000000000000000000000000100011010

olur ki bu $2^8 + 2^4 + 2^3 + 2^1 = 282$ sayısına karşılık gelir.

Küme İmplementasyonu

Aşağıdaki kod, bir `int x` değişkeni oluşturur ve bu $\{0, 1, 2, \dots, 31\}$ kümesinin herhangi bir alt kümesini tutabilir. Bundan sonra kod 1, 3, 4 ve 8 elemanlarını kümeye ekler ve kümenin büyüklüğünü yazdırır.

```
int x = 0;
x |= (1<<1);
x |= (1<<3);
x |= (1<<4);
x |= (1<<8);
cout << __builtin_popcount(x) << "\n"; // 4
```

Bundan sonra aşağıdaki kod kümeye ait olan elemanları yazdırır:

```
for (int i = 0; i < 32; i++) {
    if (x&(1<<i)) cout << i << " ";
}
// output: 1 3 4 8
```

Küme Operasyonları

Küme operasyonları bit operasyonları olarak aşağıdaki şekilde koda geçirilebilir:

	küme işareti	bit işareti
kesişim	$a \cap b$	$a \& b$
birleşim	$a \cup b$	$a b$
tamlayan	\bar{a}	$\sim a$
fark	$a \setminus b$	$a \& (\sim b)$

Örneğin, aşağıdaki kod ilk başta $x = \{1, 3, 4, 8\}$ ve $y = \{3, 6, 8, 9\}$ kümelerini oluşturur ve daha sonra $z = x \cup y = \{1, 3, 4, 6, 8, 9\}$ kümesini oluşturur:

```
int x = (1<<1)|(1<<3)|(1<<4)|(1<<8);
int y = (1<<3)|(1<<6)|(1<<8)|(1<<9);
int z = x|y;
cout << __builtin_popcount(z) << "\n"; // 6
```

Altkümelerden Geçmek

Aşağıdaki kod $\{0, 1, \dots, n-1\}$ kümesinin alt kümelerinden geçer:

```
for (int b = 0; b < (1<<n); b++) {
    // b alt kümesini şile
}
```

Aşağıdaki kod tam k elemandan oluşan alt kümelerden geçer:

```
for (int b = 0; b < (1<<n); b++) {
    if (__builtin_popcount(b) == k) {
        // b alt kümesini şile
    }
}
```

Aşağıdaki kod x kümesinin altkümelerinden geçer:

```
int b = 0;
do {
    // b alt kümesini şile
} while (b=(b-x)&x);
```

10.4 Bit Optimizasyonu

Çoğu algoritma bit operasyonları ile optimize edilebilir. Bu tip optimizasyonlar algoritmanın zaman karmaşıklığını değiştirmez fakat kodun asıl çalışma zamanında büyük bir etkisi olabilir. Bu bölümde bu tip durumlardan bahsedeceğiz.

Hamming Mesafeleri

Bir **Hamming mesafesi**, $\text{hamming}(a, b)$, eşit uzunluktaki a ve b yazılarının farklı olduğu pozisyon sayısını verir. Örneğin

$$\text{hamming}(01101, 11001) = 2.$$

Örneğin her biri k uzunluktan oluşan n bitlik yazılardan oluşan bir liste verildiğimizi düşünelim ve amacımızın listedeki iki yazı arasındaki minimum Hamming mesafesini hesaplamak olduğunu düşünelim. Örneğin $[00111, 01101, 11110]$ için cevap 2 olur çünkü

- $\text{hamming}(00111, 01101) = 2$,
- $\text{hamming}(00111, 11110) = 3$, ve
- $\text{hamming}(01101, 11110) = 3$.

Problemi çözmek için yapılan düz yollardan biri bütün yazı çiftlerinden geçip Hamming mesafesini hesaplamaktır ki bu da $O(n^2k)$ 'lık bir zaman algoritması verir. Aşağıdaki fonksiyon mesafeleri hesaplar:

```
int hamming(string a, string b) {
    int d = 0;
    for (int i = 0; i < k; i++) {
        if (a[i] != b[i]) d++;
    }
    return d;
}
```

Fakat, eğer k küçükse, kodu optimize etmek için bit yazılarını tam sayı olarak tutup Hamming mesafelerini bit operasyonları ile hesaplayabiliriz. Spesifik olarak eğer $k \leq 32$ ise yazıları `int` değerleri olarak tutup aşağıdaki fonksiyonu mesafeleri hesaplamak için kullanabiliriz:

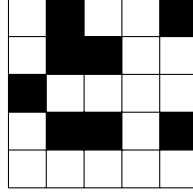
```
int hamming(int a, int b) {
    return __builtin_popcount(a^b);
}
```

Yukarıdaki fonksiyonda xor operasyonu a ve b bitlerinin farklı olduğu pozisyonlarda 1 biti bulunan bir bit yazısı oluşturur. Sonra bit sayısı `__builtin_popcount` fonksiyonu ile hesaplanır.

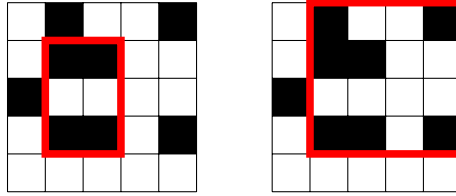
Implementasyonları karşılaştırmak için 30 uzunluğundaki 10000 random bit yazısından oluşan bir liste oluşturduk. İlk yöntemi kullanarak problemi 13.5 saniyede çözerken bit optimizasyonu ile 0.5 saniyede çözebiliriz. Böylece bit optimize edilmiş kod neredeyse orijinal koddan 30 kat daha hızlıdır.

Altılgaraları Saymak

Başka örnek olarak aşağıdaki problemi düşünelim: $n \times n$ 'lik bir ızgaranın verildiğini ve her karenin ya siyah (1) ya da beyaz (0) olduğunu varsayalım. Amacımız bütün köşeleri siyah olan altılgaraları saymaktır. Örneğin ızgara



iki altılgara içerir:



Problemi çözmek için $O(n^3)$ bir algoritma vardır: Bütün $O(n^2)$ satır çiftlerinden geçip her (a, b) çifti için her iki satırda siyah kare içeren sütun sayısını hesaplamak $O(n)$ zaman alır. Aşağıdaki kod, `color[y][x]` ifadesinin y satırında ve x sütununda bulunan karenin rengini gösterir:

```
int count = 0;
for (int i = 0; i < n; i++) {
    if (color[a][i] == 1 && color[b][i] == 1) count++;
}
```

Sonra bu sütunlar, $\text{count}(\text{count} - 1)/2$ siyah köşeli altılgaraları tutar çünkü herhangi iki tanesini bir altılgara oluşturmak için seçebiliriz.

Algoritmayı optimize etmek için ızgarayı, her biri N tane ardışık sütundan oluşan bloklara bölebilir. Sonra her satır, N -bitten oluşup karelerin rengini gösteren sayılardan oluşan bir liste tutar. Şimdi N sütunu aynı anda bit operasyonları ile işleyebiliriz. Aşağıdaki kodda, `color[y][k]`, N renkten oluşan bir bloğu bit şeklinde gösterir.

```
int count = 0;
for (int i = 0; i <= n/N; i++) {
    count += __builtin_popcount(color[a][i]&color[b][i]);
}
```

Sonuç algoritma $O(n^3/N)$ zamanda çalışır.

2500×2500 büyüklüğündeki bir rastgele ızgara oluşturduk ve orijinal ile bit optimizeli implementasyonları karşılaştırdık. Orijinal kod 29.6 saniyede çalışırken bit optimizeli versiyon $N = 32$ (int sayılar) için 3.1 saniye ve $N = 64$ (long long sayılar) için 1.7 saniye alır.

10.5 Dinamik Programlama

Bit operasyonları, elemanların alt kümelerini içeren durumları bulunduran dinamik programlama algoritmalarını verimli ve rahat bir şekilde yapmamızı sağlar çünkü bu durumları sayı olarak tutabiliriz. Şimdi bit operasyonları ile dinamik programlamayı birleştirdiğimizi örneklere bakacağız.

Optimal Seçim

İlk şu problemi düşünelim: k tane ürünün n gündeki fiyatlarının verildiğini ve her ürünü tam bir defa almamız gerektiğini düşünelim. Fakat bir günde en fazla bir tane ürün alabiliriz. Minimum toplam ücret ne olur? Örneğin aşağıdaki senaryoyu düşünelim ($k = 3$ ve $n = 8$):

	0	1	2	3	4	5	6	7
ürün 0	6	9	5	2	8	9	1	6
ürün 1	8	2	6	2	7	5	7	2
ürün 2	5	3	9	7	3	5	1	4

Bu durumda minimum toplam ücret **5** olur:

	0	1	2	3	4	5	6	7
ürün 0	6	9	5	2	8	9	1	6
ürün 1	8	2	6	2	7	5	7	2
ürün 2	5	3	9	7	3	5	1	4

Diyelim ki $\text{price}[x][d]$, k ürününün d . gündeki fiyatını versin. Örneğin yukarıdaki durumda $\text{price}[2][3] = 7$ olur. $\text{total}(S, d)$ ise de S alt kümesinde bulunan ürünleri d gününe kadar almayı sağlayacak minimum toplam ücretini verir. Bu fonksiyonu kullanarak problemin çözümü $\text{total}(\{0 \dots k-1\}, n-1)$.

İlk $\text{total}(\emptyset, d) = 0$ olur çünkü boş bir kümeyi almak para gerektirmez. $\text{total}(\{x\}, 0) = \text{price}[x][0]$ olur çünkü ilk gün bir ürünü almanın bir yolu vardır. Sonra aşağıdaki formül kullanılabilir:

$$\text{total}(S, d) = \min(\text{total}(S, d-1), \min_{x \in S} (\text{total}(S \setminus x, d-1) + \text{price}[x][d]))$$

Bu, d gününde ya ürün almayacağımızı ya da S 'e ait olan bir x ürünü alacağımızı gösterir. Sonrasında, S 'den x 'e çıkarabiliriz ve x 'in ücretini toplam ücrete ekleriz.

Sonraki adım ise fonksiyonun değerlerini dinamik programlama ile hesaplamaktır. Fonksiyon değerlerini kaydetmek için

```
int total[1<<K][N];
```

dizisini tanımlarız ve K ile N burada yeterince büyük sabitlerdir. Dizinin ilk boyutu, bir alt kümenin bit gösterimine karşılık gelir.

İlk başta $d = 0$ durumları aşağıdaki gibi işlenebilir:


```
for (int x = 0; x < k; x++) {
    total[1<<x][0] = price[x][0];
}
```

Sonra formül aşağıdaki koda dönüşür:

```
for (int d = 1; d < n; d++) {
    for (int s = 0; s < (1<<k); s++) {
        total[s][d] = total[s][d-1];
        for (int x = 0; x < k; x++) {
            if (s&(1<<x)) {
                total[s][d] = min(total[s][d],
                                   total[s^(1<<x)][d-1]+price[x][d]);
            }
        }
    }
}
```

Algoritmanın zaman karmaşıklığı $O(n2^k k)$ olur.

Permütasyonlardan Altkümelere

Dinamik programlama kullanarak permütasyonlara bakmak yerine alt kümelere bakabiliriz¹. Bunun yararı, $n!$ yani permütasyon sayısı 2^n 'den yani altküme sayısından çok daha fazladır. Örneğin $n = 20$ ise $n! \approx 2.4 \cdot 10^{18}$ ve $2^n \approx 10^6$ olur. Bu yüzden belirli n değerleri için verimli bir şekilde permütasyonlar yerine altkümelerden geçeriz.

Örnek olarak şu problemi düşünelim: Maksimum ağırlığı x olan bir asansör ve en alt kattan en üst kata çıkmak isteyen ağırlıkları bilinen n tane insan olduğunu düşünelim. Eğer insanlar asansöre optimal sırada binerlerse her insanın yukarıya çıkması için gereken en az tur sayısı kaçtır?

Örneğin $x = 10$ ve $n = 5$ olduğunu ve ağırlıkların aşağıdaki gibi olduğunu düşünelim:

kişi	ağırlık
0	2
1	3
2	3
3	5
4	6

Bu durumda, minimum tur sayısı 2 olur. Bir optimal sıra **{0,2,3,1,4}** olur ve insanları şu şekilde iki tura ayırabiliriz: ilk **{0,2,3}** (10 ağırlığında) ve sonra **{1,4}** (9 ağırlığında) olur.

Bu problem kolaylıkla $O(n!n)$ zamanda bütün n insanın olası permütasyonlarını test ederek çözmek kolaydır. Fakat dinamik programlama ile daha verimli

¹Bu teknik 1962 yılında M. Held ve R. M. Karp tarafından bulunmuştur [34].

bir $O(2^n n)$ zaman algoritması yapabiliriz. Buradaki fikir, her insanlardan oluşan altküme için iki değer hesaplamaktır: minimum gereken yol sayısı ve son grupta giden insanların minimum ağırlığı.

$\text{weight}[p]$, p insanının ağırlığını versin. İki fonksiyon tanımlarız: $\text{rides}(S)$, S alt kümesi için gereken minimum tur sayısını verir ve $\text{last}(S)$ ise de son turun minimum ağırlığını verir. Örneğin yukarıdaki senaryoda

$$\text{rides}(\{1,3,4\}) = 2x! \text{ and } x! \text{ last}(\{1,3,4\}) = 5$$

çünkü optimal turlar $\{1,4\}$ ve $\{3\}$ olur ve ikinci turun ağırlığı 5 olur. Tabii son hedefimiz $\text{rides}(\{0 \dots n-1\})$ değerini hesaplamaktır.

Fonksiyonun değerleri özyinelemeli bir şekilde hesaplayabiliriz ve dinamik programlamayı uygulayabiliriz. Buradaki fikir S grubuna ait bütün insanlardan geçip optimal bir şekilde asansöre giren son p insanını seçmektir. Her seçim daha küçük insan altkümesi için bir altproblem verir. Eğer $\text{last}(S \setminus p) + \text{weight}[p] \leq x$ ise p 'yi son tura ekleyebiliriz. Yoksa p 'yi içeren yeni bir tur oluşturmamız gerekir.

Dinamik programlamayı koda geçirmek için her S alt kümesi için bir $(\text{rides}(S), \text{last}(S))$ çifti tutan bir

```
pair<int,int> best[1<<N];
```

dizi oluştururuz. Bir boş grup için değeri şöyle belirleriz:

```
best[0] = {1,0};
```

Sonra diziyi şekildeki gibi doldurabiliriz:

```
for (int s = 1; s < (1<<n); s++) {
    // ilk değer: n+1 tur gerekir
    best[s] = {n+1,0};
    for (int p = 0; p < n; p++) {
        if (s & (1<<p)) {
            auto option = best[s^(1<<p)];
            if (option.second+weight[p] <= x) {
                // p'yi var olan bir tura ekle
                option.second += weight[p];
            } else {
                // p için yeni bir tur şoludur
                option.first++;
                option.second = weight[p];
            }
            best[s] = min(best[s], option);
        }
    }
}
```

Bu arada yukarıdaki döngü, herhangi iki S_1 ve S_2 altkümeleri için $S_1 \subset S_2$ ise S_1 'i S_2 'den daha önce işleyeceğinin garantiler. Böylece dinamik programlama değerleri doğru sırada hesaplanır.

Altkümeleri Saymak

Bölümdeki son problemimiz şöyle: $X = \{0 \dots n - 1\}$ olsun ve her $S \subset X$ altkümesi bir $\text{value}[S]$ tamsayısına tanımlanır. Amacımız, her S için

$$\text{sum}(S) = \sum_{A \subset S} \text{value}[A]$$

değerini yani S 'in altkümelerinin değerlerinin toplamını hesaplamamız gerekir.

Örneğin $n = 3$ ve değerler aşağıdaki gibi olsun:

- $\text{value}[\emptyset] = 3$
- $\text{value}[\{0\}] = 1$
- $\text{value}[\{1\}] = 4$
- $\text{value}[\{0, 1\}] = 5$
- $\text{value}[\{2\}] = 5$
- $\text{value}[\{0, 2\}] = 1$
- $\text{value}[\{1, 2\}] = 3$
- $\text{value}[\{0, 1, 2\}] = 3$

Bu durumda örneğin

$$\begin{aligned} \text{sum}(\{0, 2\}) &= \text{value}[\emptyset] + \text{value}[\{0\}] + \text{value}[\{2\}] + \text{value}[\{0, 2\}] \\ &= 3 + 1 + 5 + 1 = 10. \end{aligned}$$

Çünkü toplam 2^n altküme olduğu için bir olası çözüm bütün alt küme çiftlerinden $O(2^{2n})$ zamanda geçmektir. Buradaki fikir S 'ten çıkarılabilen elemanların toplamlarını sınırlandırmaktır.

$\text{partial}(S, k)$, S 'ten sadece $0 \dots k$ elemanlarının çıkarılabileceği şartıyla S 'in altkümelerinin değerlerinin toplamını verir. Örneğin

$$\text{partial}(\{0, 2\}, 1) = \text{value}[\{2\}] + \text{value}[\{0, 2\}],$$

çünkü sadece $0 \dots 1$ elemanlarını çıkarırız. sum değerlerini partial değerlerini kullanarak hesaplayabiliriz çünkü

$$\text{sum}(S) = \text{partial}(S, n - 1).$$

Fonksiyonun temel durumları

$$\text{partial}(S, -1) = \text{value}[S]$$

olur çünkü bu durumda S 'ten herhangi bir eleman çıkarılamaz. Sonra genel durumda aşağıdaki formülü kullanabiliriz:

$$\text{partial}(S, k) = \begin{cases} \text{partial}(S, k - 1) & k \notin S \\ \text{partial}(S, k - 1) + \text{partial}(S \setminus \{k\}, k - 1) & k \in S \end{cases}$$

Şimdi burada k elemanına odaklanalım. Eğer $k \in S$ ise iki seçeneğimiz var: ya S 'te k 'yi tutarız ya da S 'ten çıkarırız.

Burada toplamaları hesaplamamanın zeki bir yolu vardır. Bunun için

```
int sum[1<<N];
```

dizisini hesaplarız ki bu her altkümenin toplamını verir. Dizi aşağıdaki gibi tanımlanır:

```
for (int s = 0; s < (1<<n); s++) {  
    sum[s] = value[s];  
}
```

Sonra diziyi şekildeki doldururuz:

```
for (int k = 0; k < n; k++) {  
    for (int s = 0; s < (1<<n); s++) {  
        if (s&(1<<k)) sum[s] += sum[s^(1<<k)];  
    }  
}
```

Bu kod, $k = 0 \dots n - 1$ kadarki $\text{partial}(S, k)$ değerlerini sum dizisine hesaplarız. Şimdi $\text{partial}(S, k)$ her zaman $\text{partial}(S, k - 1)$ değerine bağlı olduğu için sum dizisini yeniden kullanabiliriz ki bu da verimli bir implementasyon verir.

Kısım II

Çizge Algoritmaları

Bölüm 11

Çizgenin Temelleri

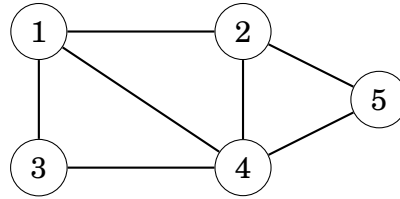
Çoğu kodlama sorusu problemi çizge problemi gibi resmedip uygun bir çizge algoritması ile çözülebilir. Tipi bir çizge örneği, bir ülkedeki yolların ve şehirlerin oluşturduğu ağıdır. Bazen sorudaki çizge saklı olabileceği için fark edilmesi zordur.

Kitabın bu kısmı çizge algoritmalarını anlatacaktır ve özellikle rekabetçi programlamada önemli olan kısımlara odaklanacaktır. Bu bölümde çizgelerle alakalı konseptleri konuşup algoritmalarda çizgeleri göstermenin farklı yollarını çalışacağız.

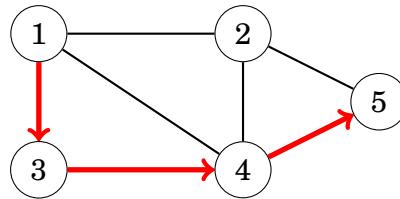
11.1 Çizge Terminolojisi (Graph Terminology)

Bir **çizge** (graph), **düğüm**lerden (nodes) ve **kenarlardan** (edges) oluşur. Bu kitapta n çizgedeki toplam düğüm sayısını ve m toplam kenar sayısını verecektir. Düğümler $1, 2, \dots, n$ tamsayılarını kullanarak numaralandırılacaktır.

Örneğin aşağıdaki çizge, 5 düğüm ve 7 kenardan oluşur:



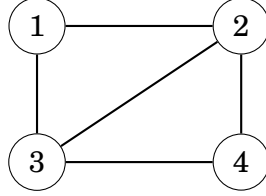
Yol (path), a düğümünü b düğüme kenarlardan geçerek varmasını sağlar. Yolun **uzunluğu**, yolda geçtiğimiz kenar sayısına eşittir. Örneğin yukarıdaki çizgede $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$ yolu 1. düğümden 5. düğüme 3 uzunluğunda bir yoldur.



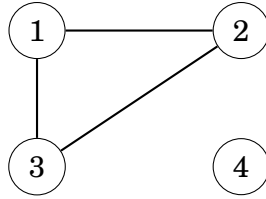
Eğer bir yolda başlangıç ile son düğüm aynı ise bu yol bir **döngüdür** (cycle). Örneğin yukarıdaki çizge $1 \rightarrow 3 \rightarrow 4 \rightarrow 1$ düğümünü içerir. Eğer bir yolda her düğüm en fazla bir defa bulunuyorsa bu yol **basittir** (simple).

Bağlılık (Connectivity)

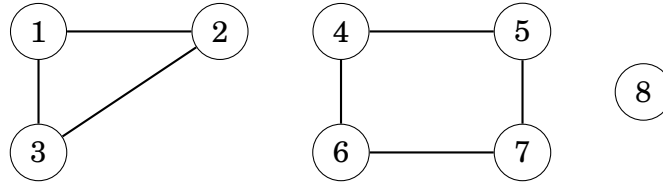
Eğer her iki düğümü arasında bir yol varsa bu çizge bağlıdır (connected). Örneğin aşağıdaki çizge bağlıdır:



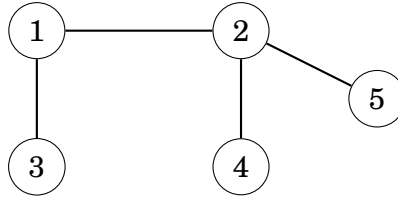
Aşağıdaki çizge bağlı değildir çünkü 4. düğümden diğer hiçbir düğüme gidilemez:



Bir çizgenin bağlı gruplarına **parça** (component) denir. Örneğin aşağıdaki çizgenin $\{1, 2, 3\}$, $\{4, 5, 6, 7\}$ ve $\{8\}$ olmak üzere üç tane parçası vardır:

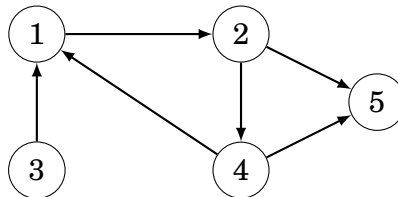


Eğer bir çizge bağlıysa ve n tane düğümden ve $n - 1$ kenardan oluşuyorsa bu çizge bir **ağaçtır** (tree). Bir ağaçta her iki düğüm arasında farklı bir yol vardır. Örneğin aşağıdaki çizge bir ağaçtır:



Kenar Yönleri (Edge Directions)

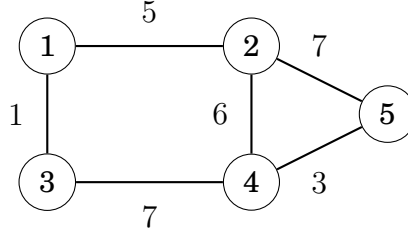
Eğer bir çizgenin kenarları tek yönlüyse bu çizge **yönlüdür**. Örneğin aşağıdaki çizge yönlüdür:



Yukarıdaki çizge 3. düğümden 5. düğüme $3 \rightarrow 1 \rightarrow 2 \rightarrow 5$ yolunu içerir ama 5. düğümden 3. düğüme herhangi bir yol yoktur.

Kenar Ağırlıkları (Edge Weights)

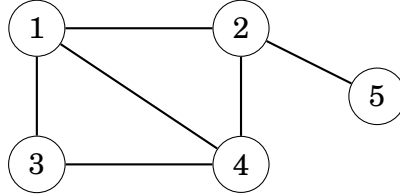
Ağırlıklı bir çizgede, her kenarın bir **ağırlığı** vardır. Bu ağırlıklar genel olarak kenar uzunluğu olarak söylenir. Örneğin aşağıdaki çizge ağırlıklıdır:



Ağırlıklı bir çizgedeki yolun uzunluğu, o yoldaki kenarlarının ağırlıklarının toplamıdır. Örneğin yukarıdaki çizgede $1 \rightarrow 2 \rightarrow 5$ yolunun uzunluğu **12** ve $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$ yolunun uzunluğu **11**'dir. İkinci yol 1. düğümden 5. düğüme **en kısa** yoldur.

Komşular ve Dereceler (Neighbors and Degrees)

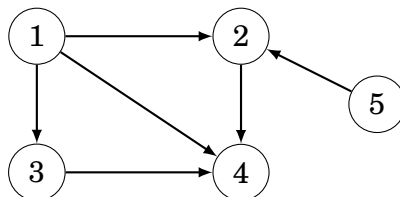
İki düğüm arasında bir kenar varsa bunlar **komşu** düğümlerdir. Bir düğümün **derecesi**, o düğümün komşu sayısına eşittir. Örneğin aşağıdaki çizgede 2. düğümün komşuları 1, 4 ve 5'tir. Bu yüzden 2. düğümün derecesi 3'tür.



m tane kenar bulunduran bir çizgenin derece toplamı her zaman **$2m$** olur çünkü bir kenar, iki tane düğümün derece sayısını 1 artırır. Bu nedenden dolayı kenar sayısı her zaman çifttir. Bu durum her iki düğüm arasında en fazla bir kenar olduğu zaman doğrudur.

Her düğümün derece sayısı sabit d ise bu çizge **sıradan** (regular) ve eğer her düğümün derece sayısı $n - 1$ ise yani bütün düğümler birbirine bağlı ise bu çizge **tam** (complete) çizgedir.

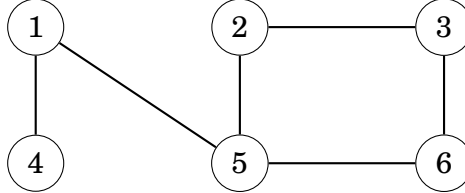
Yönlü bir çizgede bir düğümün **iç derecesi** (indegree), o düğüme gelen kenar sayısını, **dış derecesi** (outdegree) ise o düğümden çıkan kenar sayısını vermektedir. Örneğin aşağıdaki çizgede 2. düğümün iç derecesi 2 iken dış derecesi 1'dir.



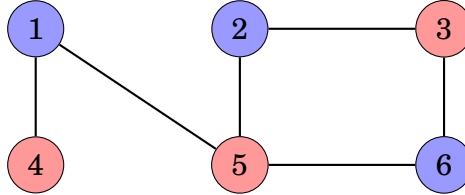
Boyamalar (Colorings)

Bir çizmeyi **boyarken** her düğümün komşularından farklı bir renge sahip olacak şekilde boyanmasına dikkat edilir.

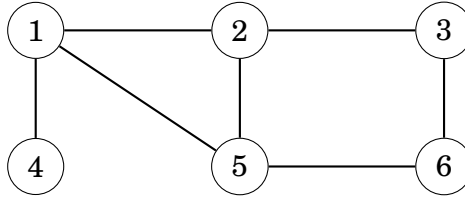
Eğer bütün çizmeyi sadece iki renkle boyamak mümkünse bu çizme iki parçalıdır (bipartite). Bir çizmenin iki parçalı olması için tek sayıda kenar içerip herhangi bir döngü bulundurmaması gerekir. Örneğin:



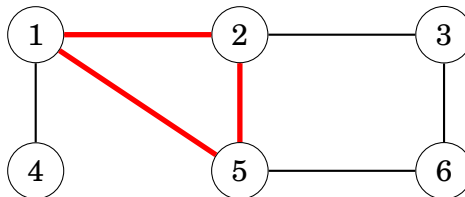
iki parçalıdır çünkü bu çizme aşağıdaki gibi boyanabilir:



Fakat

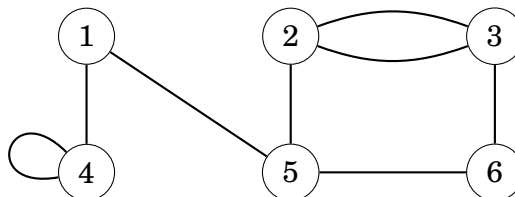


çizmesi iki parçalı değildir çünkü aşağıdaki döngüden dolayı çizme sadece iki renkle boyanamaz:



Basitlik (Simplicity)

Eğer çizme, aynı düğümde başlayıp biten bir kenar içermiyorsa bu çizme **basittir** (simple). Genelde çizmelerin basit olduğunu kabul ederiz fakat basit olmadıkları zaman da olabilmektedir. Örneğin aşağıdaki çizme basit **değildir**:



11.2 Çizge Gösterimi (Graph Representation)

Algoritmalarda çizgeler göstermenin birkaç yolu vardır. Veri yapısının seçimi çizgenin büyüklüğüne ve algoritmanın çizgeyi işleme yoluna göre değişir. Şimdi üç tane yaygın çizge gösterimine bakacağız.

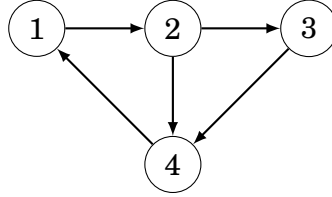
Komşuluk Listesi Gösterimi (Adjacency List Representation)

Komşuluk listesi gösteriminde, çizgedeki her x düğümü **komşuluk listesi** bir yere atanır ve bu yerde x 'den çıkan kenarların olduğu düğümler bulunur. Komşuluk listeleri çizgeleri göstermenin en popüler halidir ve çoğu algoritma bu yöntem kullanılarak verimli bir şekilde koda geçirilebilir.

Komşuluk listesini oluşturmanın rahat yollarından biri vektörlerden oluşan bir dizi oluşturmaktır:

```
vector<int> adj[N];
```

N sabiti, bütün komşuluk listelerinin kaydedilebilmesi için yeterince büyük bir şekilde seçilmiştir. Örneğin



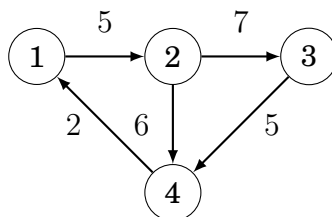
çizgesi aşağıdaki gibi kaydedilebilir:

```
adj[1].push_back(2);  
adj[2].push_back(3);  
adj[2].push_back(4);  
adj[3].push_back(4);  
adj[4].push_back(1);
```

Eğer çizge yönsüzse (undirected), her kenar iki tarafa da eklenecek şekilde yukarıdaki gibi tutulabilir. Ağırlıklı bir çizgede yapı aşağıdaki gibi güncellenebilir:

```
vector<pair<int,int>> adj[N];
```

Bu durumda, eğer a düğümünden b düğüme w ağırlığında bir kenar varsa a düğümünün komşuluk listesi (b,w) çiftini içerir. Örneğin



çizgesi aşağıdaki şekilde kaydedilebilir:

```
adj[1].push_back({2,5});
adj[2].push_back({3,7});
adj[2].push_back({4,6});
adj[3].push_back({4,5});
adj[4].push_back({1,2});
```

Komşuluk listesini kullanmanın yararı bir düğümden kenarla gidilebilecek düğümleri hızlı bir şekilde bulabilmemizdir. Örneğin aşağıdaki döngü s düğümünden gidebileceğimiz bütün düğümleri gösterir:

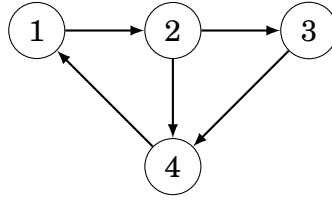
```
for (auto u : adj[s]) {
    // u düğümünü uygula
}
```

Komşuluk Matrisi Gösterimi (Adjacency Matrix Representation)

Komşuluk matrisi iki boyutlu bir dizi olup çizgenin içerdiği kenarları gösterir. Komşuluk matrisi iki tane köşe arasında kenar bulunup bulunmadığını hızlı bir şekilde bulabiliriz. Matris dizi olarak:

```
int adj[N][N];
```

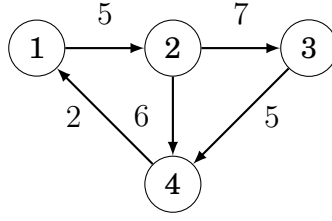
şeklinde tutulabilir. Her $adj[a][b]$ değeri, a düğümünden b düğüme kenar bulunup bulunmadığını söyler. Eğer çizgede bu kenar varsa $adj[a][b] = 1$ ve eğer yoksa $adj[a][b] = 0$ olur. Örneğin



çizgesi aşağıdaki şekilde gösterilebilir:

	1	2	3	4
1	0	1	0	0
2	0	0	1	1
3	0	0	0	1
4	1	0	0	0

Eğer çizge ağırlıklı ise komşuluk matrisi kenarın ağırlığını da içerecek şekilde güncellenebilir. Bu gösterimi kullanarak



çizgesi aşağıdaki matris ile gösterilebilir:

	1	2	3	4
1	0	5	0	0
2	0	0	7	6
3	0	0	0	5
4	2	0	0	0

Komşuluk matrisinin dezavantajı matrisin çoğu sıfır olan n^2 eleman bulundurmasıdır. Bu yüzden, çizge çok büyükse bu gösterim kullanılamaz.

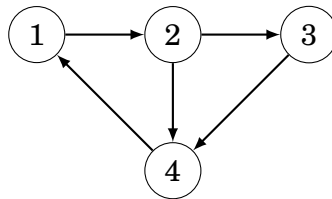
Kenar Listesi Gösterimi (Edge List Representation)

Kenar listesi bir çizgenin bütün kenarların rastgele bir sırada tutar. Eğer algoritma çizgenin bütün kenarlarını işliyorsa ve verilen bir düğümden başlayan kenarları bulmaya gerek yoksa bu gösterim işe yarardır.

Kenar listesi vektörde tutulabilir:

```
vector<pair<int,int>> edges;
```

her (a,b) çifti a düğümünden b düğüme kenar olduğunu gösterir. Böylece



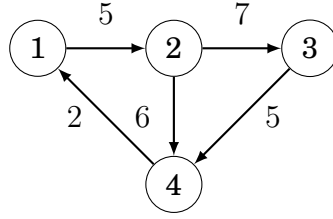
çizgesi aşağıdaki şekilde gösterilebilir:

```
edges.push_back({1,2});
edges.push_back({2,3});
edges.push_back({2,4});
edges.push_back({3,4});
edges.push_back({4,1});
```

Eğer çizge ağırlıklıysa, yapı aşağıdaki gibi güncellenebilir:

```
vector<tuple<int,int,int>> edges;
```

Listedeki her eleman (a,b,w) formunda olup a düğümünden b düğümüne w ağırlığında bir kenar olduğundan bahseder. Örneğin



çizgesi aşağıdaki şekilde gösterilebilir¹:

```
edges.push_back({1,2,5});  
edges.push_back({2,3,7});  
edges.push_back({2,4,6});  
edges.push_back({3,4,5});  
edges.push_back({4,1,2});
```

¹Bazı eski derleyicilerde süslü parantez yerine `make_tuple` fonksiyonu kullanılmalıdır. Örneğin `{1,2,5}` yerine `make_tuple(1,2,5)` kullanılmalıdır.

Bölüm 12

Çizgede Dolaşma (Graph Traversal)

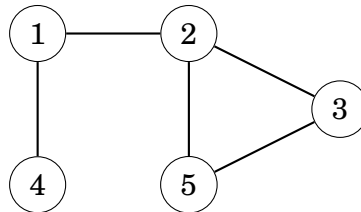
Bu bölüm iki temel çizge algoritmasından bahsedecek: Derinlik öncelikli arama ve genişlik öncelikli arama. İki algoritmanın da başlangıç noktası vardır ve ikisi de başlangıç düğümünden varılabilecek bütün düğümleri dolaşırlar. İkisinin farkı, düğümleri dolaşma sırasıdır.

12.1 Derinlik Öncelikli Arama (Depth-First Search)

Derinlik öncelikli arama (DFS) düz bir çizge dolaşma tekniğidir. Algoritma başlangıç düğümünden başlayıp çizgenin kenarlarını kullanarak bütün ulaşılabilir düğümleri gezer. Derinlik öncelikli arama her zaman yeni düğüm bulunduğu sürece tek bir yolu takip eder. Bundan sonra önceki ziyaret ettiği düğümlere dönüp çizgenin diğer kısımlarını ziyaret eder. Algoritma her dolaşılan düğümü kaydeder ve bu yüzden her düğüm sadece bir defa ziyaret edilip işlenir.

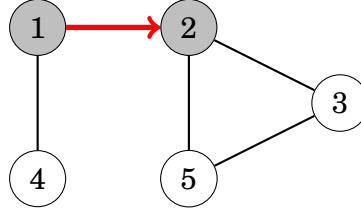
Örnek

Derinlik öncelikli aramanın aşağıdaki çizgeyi nasıl dolaşacağını düşünelim:

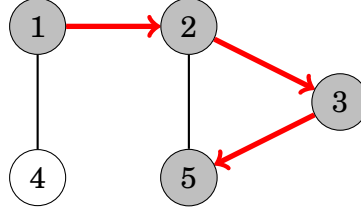


Aramaya çizgedeki herhangi bir düğümünden başlayabiliriz. Şimdi aramaya 1. düğümünden başlayacağız.

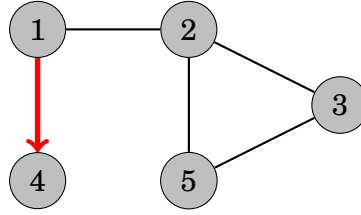
Arama ilk 2. düğüme ilerler:



Sonra 3 ve 5. düğümler ziyaret edilir:



5. düğümün komşuları 2 ve 3'tür fakat algoritma ikisini de önceden ziyaret ettiği için önceki düğümlere geri döner. 3 ve 2. düğümlerinin de bütün kenarları ziyaret edildiği için 1. düğüme döner. Sonrasında 1. düğümden 4. düğüme gider:



Bundan sonra bütün düğümler ziyaret edildiği için arama sonlanır.

Derinlik öncelikli aramanın zaman karmaşıklığı $O(n + m)$ olup n düğüm sayısını m ise kenar sayısını belirtir. Zaman karmaşıklığının böyle olmasının sebebi algoritmanın her düğümü ve kenarı bir kere işlemesidir.

Implementasyon (Implementation)

Derinlik öncelikli arama özyineleme (recursion) ile koda dökülebilir. `dfs` fonksiyonu belirtilen düğümden derinlik öncelikli aramaya başlar. Fonksiyon, çizgenin dizi içinde komşuluk listeleri ile tutulduğunu varsayar:

```
vector<int> adj[N];
```

ve aynı zamanda

```
bool visited[N];
```

adında ziyaret edilen düğümleri tutan bir dizi tutar. Başta dizideki her değer `false` olup arama s düğüme geldiği zaman `visited[s]` `true` olur. Fonksiyon aşağıdaki şekilde koda geçirilebilir:

```
void dfs(int s) {
    if (visited[s]) return;
```



```
visited[s] = true;
// s düğümüyle bir şeyler yap
for (auto u: adj[s]) {
    dfs(u);
}
}
```

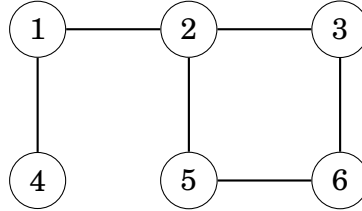
12.2 Genişlik Öncelikli Arama (Breadth-First Search)

Genişlik öncelikli arama (BFS) düğümleri başlangıç düğümüne olan mesafelere göre artan bir şekilde ziyaret eder. Bu yüzden genişlik öncelikli arama kullanarak bütün düğümlerin başlangıç düğümüne olan mesafeleri bulabiliriz. Fakat, genişlik öncelikli arama, derinlik öncelikli aramaya göre koda geçirilmesi daha zordur.

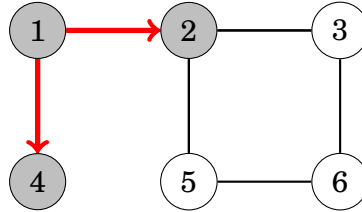
Genişlik öncelikli arama düğümleri level level dolaşır. İlk başta başlangıç düğümüne mesafesi 1 olan düğümleri ziyaret eder, sonra mesafesi 2 olanları ve böyle devam eder. Bu işlem bütün düğümler ziyaret edilene kadar devam eder.

Örnek

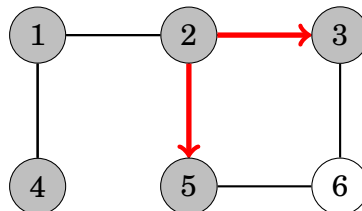
Genişlik öncelikli aramanın aşağıdaki çizgeyi nasıl ziyaret ettiğine bakalım:



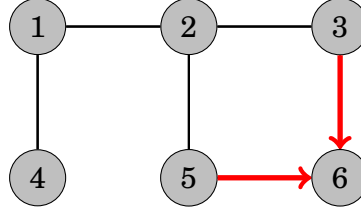
Aramanın 1. düğümünden başladığını varsayalım. İlk başta 1. düğümünden tek bir kenar kullanılarak varılacak bütün düğümleri ziyaret ederiz:



Sonra 3 ve 5. düğümleri ziyaret ederiz:



En sonda 6. düğümü ziyaret ederiz:



Böylece başlangıç düğümünden bütün düğümlere olan mesafeleri hesaplamış olduk. Mesafeler aşağıdaki gibi olur:

düğüm	mesafe
1	0
2	1
3	2
4	1
5	2
6	3

Derinlik öncelikli aramadaki gibi genişlik öncelikli aramanın zaman karmaşıklığı $O(n + m)$ olup n düğüm sayısını m ise kenar sayısını belirtir.

Implementasyon

Genişlik öncelikli aramanın koda geçirilmesi derinlik öncelikli aramaya göre daha zordur çünkü algoritma çizgenin farklı yerlerindeki düğümleri ziyaret eder. Klasik bir implementasyon kuyruk (queue) yapısını kullanır. Her adımda kuyruksa ilk sırada bulunan düğüm işlenir.

Aşağıdaki kod çizgenin komşuluk listelerinde tutulduğunu varsayar ve aşağıdaki veri yapılarını tutar:

```
queue<int> q;  
bool visited[N];  
int distance[N];
```

`q` kuyruğu düğümleri artan mesafeye göre işlemlerini sağlar. Yeni düğümler her zaman en sona eklenir ve baştaki düğümse sonraki işlenecek düğüm olur. `visited` dizisi aramada ziyaret edilmiş düğümleri gösterir ve `distance` dizisi ise başlangıç düğümünden diğer düğümlere olan mesafeleri tutar.

Arama, x düğümünü başlangıç düğümü olarak alarak aşağıdaki gibi koda geçirilebilir:

```
visited[x] = true;  
distance[x] = 0;  
q.push(x);  
while (!q.empty()) {  
    int s = q.front(); q.pop();  
    // s düğümüyle bir şeyler yap
```

```

for (auto u : adj[s]) {
    if (visited[u]) continue;
    visited[u] = true;
    distance[u] = distance[s]+1;
    q.push(u);
}
}

```

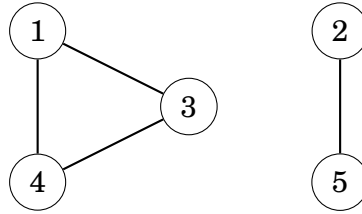
12.3 Uygulamalar

Çizge dolaşma algoritmalarını kullanarak çizgenin çeşitli özelliklerini kontrol edebiliriz. Genelde derinlik öncelikli arama ile genişlik öncelikli aramanın ikisi de kullanılabilir ama daha kolay kodu yazıldığı için pratikte derinlik öncelikli arama daha iyi bir seçenektir. Aşağıdaki uygulamalar, çizgenin yönsüz olduğunu kabul eder.

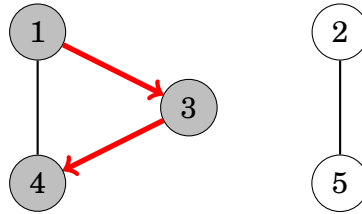
Bağlılık Kontrolü

Eğer çizgenin her iki düğümü arasında yol varsa bu çizge bağlıdır. Böylece çizgenin bağlı olup olmadığını bir tane düğümden diğer bütün düğümlere ulaşıp ulaşılmadığını kontrol ederek bulabiliriz.

Örneğin



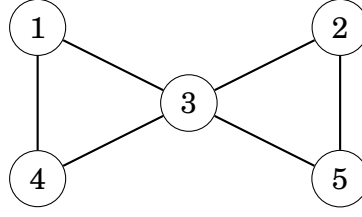
çizgesinde 1. düğümden başlatılarak yapılan derinlik öncelikli arama aşağıdaki düğümleri ziyaret eder:



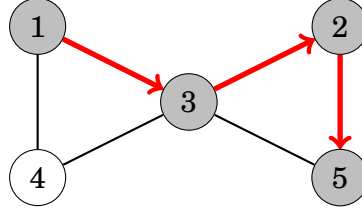
Arama bütün düğümleri ziyaret etmediği için çizgenin bağlı olmadığını kabul ederiz. Benzer şekilde, çizgenin bütün bağlı parçalarını her seferinde bir parçaya ait olmayan düğümden arama yaparak bulabiliriz:

Döngü Bulmak

Eğer çizgeyi dolaşırken önceden ziyaret ettiğimiz bir komşu düğüm bulursak (şu anki olduğumuz düğümümüzden bir önceki olduğumuz düğüm hariç) bu çizgede döngü olduğunu bulmuş oluruz. Örneğin



iki döngü içerir ve bir tanesini aşağıdaki gibi bulabiliriz:



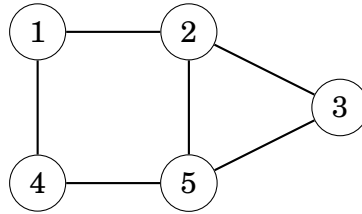
2. düğümden 5. düğüme giderken 5. düğümün komşularından olan 3. düğümün zaten ziyaret edildiğini fark ederiz. Bu yüzden 3. düğümden giden bir $3 \rightarrow 2 \rightarrow 5 \rightarrow 3$ döngüsü bulmuş oluruz.

Döngü bulmanın başka bir yolu ise çizgedeki her parçanın düğüm ve kenar sayısını hesaplamaktır. Eğer parça c tane düğüm içeriyorsa ve hiç döngü bulundurmuyorsa $c - 1$ kenar içerir (yani bu bir ağaçtır.). Eğer c veya daha fazla kenar bulunduruyorsa, bu parça bir döngü içermektedir.

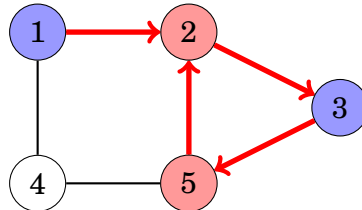
İki Parçalılık Kontrolü (Bipartiteness Check)

Eğer komşu düğümlerde aynı renk olmayacak şekilde çizgeyi sadece iki renk ile boyayabiliyorsak bu çizge iki parçalıdır (bipartite). Bir çizgenin iki parçalı olup olmadığını çizge dolaşma algoritmaları ile çok kolay bir şekilde bulabiliriz. Buradaki fikir, başlangıç düğümünü maviye boyayıp diğer bütün komşularını kırmızıya boyamaktır. Sonrasında bu kırmızı düğümlerin diğer komşularını maviye boyayarak devam ederiz. Eğer iki komşu düğüm aynı renge boyandıysa bu çizgenin iki parçalı olmadığını gösterir. Eğer böyle bir durum olmazsa bu çizgenin iki parçalı olduğunu gösterir ve olası boyamalardan bir tanesini bulmuş oluruz.

Örneğin



çizgesi iki parçalı düğüldür çünkü 1. düğümden yapılan bir arama aşağıdaki gibi ilerler:



Burada 2 ve 5. düğümün komşu olmalarına rağmen ikisinin de kırmızı olduğunu görürüz. Bu yüzden bu çizge iki parçalı değildir.

Bu algoritma iki tane renk olduğu zaman hep çalışır. Başlangıç düğümünün rengi diğer bütün düğümlerin rengini belirler. Başlangıç düğümünü kırmızı veya mavi yapmanın herhangi bir farkı yoktur.

Genel durumda, bir çizgenin düğümlerini komşu düğümlerin aynı renge sahip olmayacağını varsayarak k renk ile boyanıp boyanamayacağını bulmak zordur. $k = 3$ olsa dahi bu soru için bilinen verimli bir algoritma yoktur yani NP-hard'dır.

Bölüm 13

En Kısa Yolu Bulmak (Shortest Paths)

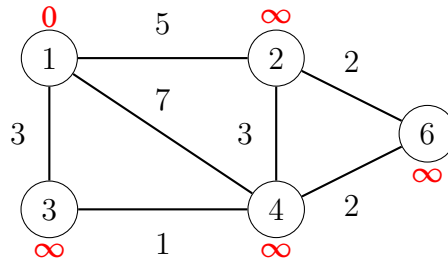
Bir çizgede iki düğüm arası en kısa yolu bulmak bir sürü pratikte uygulaması olan önemli bir problemdir. Örneğin klasik problemlerden biri, yolların uzunlukları bilinen bir yol ağında iki şehir arasındaki en kısa rotayı bulmaktır. Ağırlıksız çizgede rotanın uzunluğu kenar sayısına eşittir ve bunu genişlik öncelikli arama (BFS) ile bulabiliriz. Fakat, bu bölümde daha gelişmiş algoritmalar gerektiren ağırlıklı çizgelerde en kısa yolu bulmayı bakacağız.

13.1 Bellman–Ford Algoritması

Bellman-Ford algoritması¹ başlangıç düğümünden çizgedeki bütün düğümlere en kısa mesafeyi bulur. Algoritma negatif döngü içermediği sürece her türlü çizgede çalışabilir. Eğer çizge negatif döngü içeriyorsa algoritma bunu fark edebilir. Algoritma başlangıç düğümünden diğer bütün düğümlere olan uzaklıkları tutar. Başta başlangıç düğümünün mesafesi 0 olup diğer bütün düğümlere olan uzaklık sonsuzdur. Algoritma bu mesafeleri yolu kısaltan kenarlar bularak azaltır. Algoritma artık herhangi bir mesafeyi azaltamadığı zaman durur.

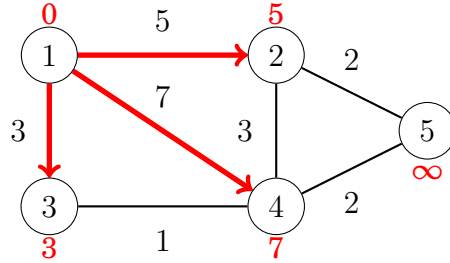
Örnek

Bellman-Ford algoritmasının aşağıdaki çizgede nasıl çalıştığını bakalım:

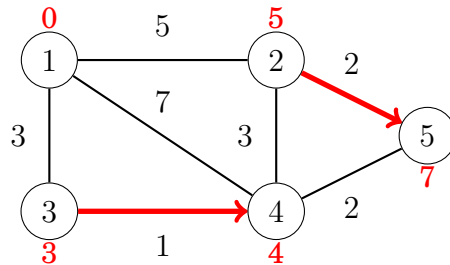


¹Algoritma R. E. Bellman ve L. R. Ford kişilerinden ismini almıştır. Bu kişiler, algoritmayı birbirlerinden habersiz bir şekilde sırasıyla 1958 ve 1956 yılında yayınlamışlardır [5, 24].

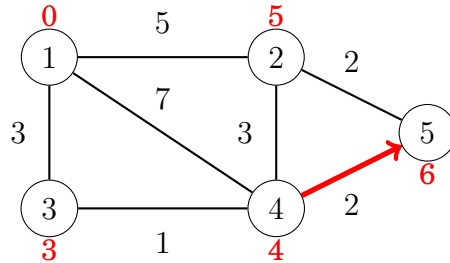
Çizgedeki her düğüme bir mesafe atanır. Başta başlangıç düğümünün mesafesi 0 olup diğer bütün düğümlerin mesafesi sonsuzdur. Algoritma mesafeleri azaltacak kenarları aramaya başlar. İlk başta 1. düğümden çıkan bütün kenarlar mesafeleri azaltır:



Bundan sonra $2 \rightarrow 5$ ve $3 \rightarrow 4$ kenarları mesafeleri azaltır:

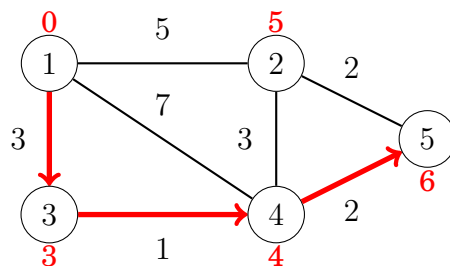


Sonda bir tane daha değişiklik olur:



Bundan sonra herhangi bir kenar mesafeyi azaltmaz. Yani bu, başarıyla başlangıç düğümünden bütün düğümlere en kısa mesafeleri bulduğumuz anlamına gelir.

Örneğin 1. düğümden 5. düğüme en kısa uzunluk 3 olup aşağıdaki gibidir:



Implementasyon

Aşağıdaki Bellman-Ford algoritmasının kodu, x düğümünden çizgenin diğer düğümlerine en kısa mesafeyi bulur. Kod, çizgenin kenar listesi olan `edges`'de tutulduğunu varsayar. Bu listede (a, b, w) şeklinde demetlerden oluşup bu a düğümünden b düğüme w ağırlığında bir kenar olduğunu gösterir.

Algoritma $n - 1$ tur çalışır ve her turda algoritma çizgenin bütün kenarlarına bakıp mesafeleri kısaltmaya çalışır. Algoritma x düğümünden diğer bütün düğümlere mesafeleri tutan `distance` adında bir dizi oluşturur. `INF` sabiti sonsuz mesafeyi belirtir.

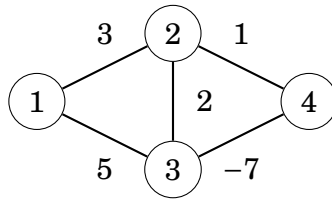
```
for (int i = 1; i <= n; i++) distance[i] = INF;
distance[x] = 0;
for (int i = 1; i <= n-1; i++) {
    for (auto e : edges) {
        int a, b, w;
        tie(a, b, w) = e;
        distance[b] = min(distance[b], distance[a]+w);
    }
}
```

Algoritmanın zaman karmaşıklığı $O(nm)$ idir çünkü algoritma $n - 1$ turdan oluşup her turda m kenarı işler. Eğer çizgenin negatif döngüsü yoksa $n - 1$ turdan sonra bütün mesafeler en kısa hale gelir çünkü her en kısa yol en fazla $n - 1$ kenar içerir.

Pratikte, son mesafeler genelde $n - 1$ turdan daha önce bulunur. Bu yüzden bir tur boyunca herhangi bir mesafe değişmezse algoritmayı durdurarak daha verimli hale getirilebilir.

Negatif Döngüler

Belmann-Ford algoritması çizge negatif döngü içerip içermediğini kontrol eder. Örneğin



çizgesi $2 \rightarrow 3 \rightarrow 4 \rightarrow 2$ uzunluğundan negatif bir döngü içerir.

Eğer çizge negatif döngü içerirse döngüyü içeren yolu her defasında döngüyü kullanarak azaltabiliriz. Bu yüzden böyle bir durumda en kısa yol mantıklı değildir.

Negatif döngü Bellman-Ford algoritması kullanarak n turda bulunabilir. Eğer son turda mesafe azalırsa çizgede negatif döngü olduğu görülür. Bu algoritma başlangıç düğümü fark etmeksizin çizgede herhangi bir negatif döngü içerip içermediği kontrol edilebilir.

SPFA Algoritması

SPFA algoritması ("Shortest Path Faster Algorithm") [20] Bellman-Ford algoritmasının bir tiği olup orijinal algoritmadan genelde daha verimlidir. SPFA algoritması her turda bütün kenarlardan geçmez. Onun yerine seçilecek kenarları daha zeki bir şekilde seçer.

Algoritma mesafeleri azaltabilecek düğümlerden oluşan bir kuyruk (queue) yapısı tutar. İlk başta, algoritma x başlangıç düğümünü kuyruğa ekler. Sonra algoritma hep kuyrukta başta olan düğümü işler ve ne zaman bir $a \rightarrow b$ kenarı mesafeyi azaltırsa o zaman b düğümü kuyruğa eklenir.

SPFA algoritmasının verimliliği çizgenin yapısına bağlıdır ama yine de en kötü durumda zaman karmaşıklığı hala $O(nm)$ olur ve bu orijinal Bellman-Ford algoritması kadar SPFA algoritmasını yavaş çalıştıracak girdiler oluşturarak mümkündür.

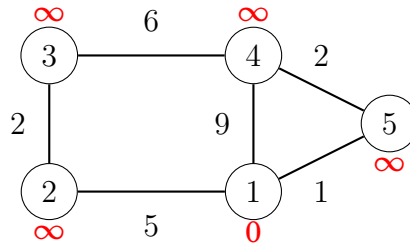
13.2 Dijkstra'nın Algoritması

Dijkstra'nın algoritması² başlangıç düğümünden çizgenin diğer tüm düğümlerine Bellman-Ford algoritması gibi en kısa mesafeyi bulur. Dijkstra'nın algoritmasının yararı ise daha verimli olup daha büyük çizgelerde kullanılabilir olmasıdır. Fakat algoritmanın çalışabilmesi için çizgede negatif ağırlıklı kenar bulunmaması gerekir.

Bellman-Ford algoritması gibi Dijkstra'nın algoritması da düğümlere olan mesafeleri tutar ve bunları arama sırasında azaltır. Dijkstra'nın algoritmasının verimli olmasının sebebi çizgedeki her kenarı bir defa işlemesidir. Bunu da sağlayan çizgede negatif ağırlıklı kenar bulunmamasıdır.

Örnek

Dijkstra'nın algoritmasının başlangıç düğümünün 1. düğüm olduğu zaman nasıl çalıştığına bakalım:

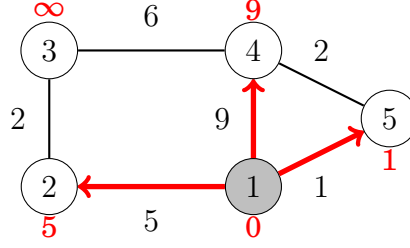


Bellman-Ford algoritması gibi başta başlangıç düğümüne olan uzaklık 0 olup diğer bütün düğümlere olan uzaklık ise sonsuzdur.

Her adımda, Dijkstra'nın algoritması hiç işlenilmemiş bir düğümü seçer ve bu düğümü seçerken de seçilmemiş düğümler arasında mesafesi en kısa olan düğümü seçmeye dikkat eder. İlk düğüm 1. düğüm olup mesafesi 0'dır.

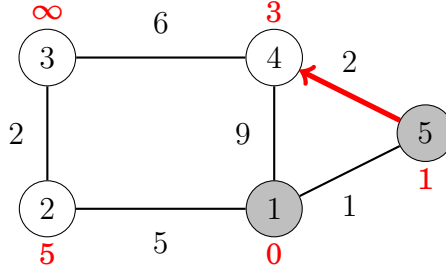
²E. W. Dijkstra bu algoritmayı 1959 yılında yayınlamıştır [14], fakat orijinal yayınladığı kağıdında algoritmanın verimli ve hızlı şekilde nasıl koda geçirileceğinden bahsetmemiştir.

Bir düğüm seçildiği zaman algoritma o düğümden başlayan bütün kenarlarından geçer ve bu kenarları kullanarak mesafeleri azaltmaya çalışır:

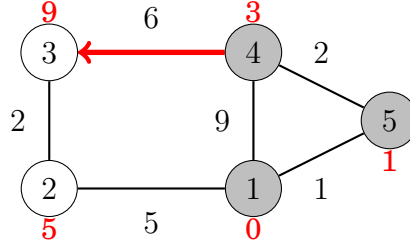


Bu durumda, 1. düğümden çıkan kenarlar 2, 4 ve 5. düğümün mesafelerini azaltarak sırasıyla 5, 9 ve 1 yapar.

Sonraki işlenecek düğüm ise 1 mesafesiyle 5. düğümdür. Böylece 4. düğümün mesafesi 9'dan 3'e azalır.

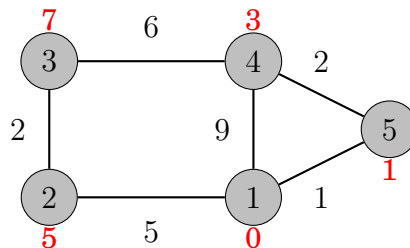


Bundan sonra, sıradaki düğüm 4. düğümdür ve bu düğüm 3. düğüme olan mesafeyi 9'a indirir:



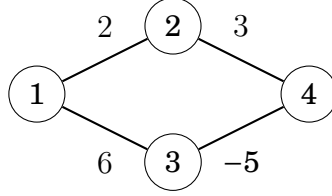
Dijkstra'nın algoritmasının önemli özelliklerinden biri, bir düğüm seçildiği zaman onun mesafesinin daha değişmiyor olmasıdır. Örneğin algoritmanın şu anında 1, 5 ve 4. düğümlerin son mesafeleri 0, 1 ve 3'tür ve daha sonra değişmez.

Bundan sonra algoritma geriye kalan iki düğümü de işler ve son mesafeler aşağıdaki gibi olur:



Negatif Kenarlar

Dijkstra'nın algoritmasının verimliliği çizgenin negatif kenar içermemesinden gelir. Eğer çizgede negatif kenar varsa algoritma yanlış sonuçlar verebilir. Örneğin



1. düğümden 4. düğüme en kısa yol $1 \rightarrow 3 \rightarrow 4$ olup mesafesi 1'dir. Fakat Dijkstra'nın algoritması en kısa kenarları alarak $1 \rightarrow 2 \rightarrow 4$ yolunu bulur. Algoritma diğer yoldaki -5 ağırlığındaki kenarın öncesindeki 6 uzunluğunu azalttığını fark etmiyor.

Implementasyon

Aşağıdaki Dijkstra'nın algoritmasının koda geçirilmiş hali x düğümünden diğer kalan bütün düğümlere en kısa mesafeleri bulur. Çizge komşuluk listelerine tutulur yani $adj[a](b, w)$ çifti içerir. Bu a düğümünden b düğüme w ağırlığında bir kenar olduğu anlamına gelir.

Dijkstra algoritmasının verimli implementasyonu için hızlı bir şekilde işlenmemiş düğümler arasında mesafesi en küçük düğümü bulmayı gerektirir. Bunu sağlayabilecek veri yapılarından biri ise öncelikli kuyruktur (priority queue). Öncelikli kuyruk düğümleri mesafelerine göre sıralar. Öncelikli kuyruk kullanarak sonraki işlenecek düğümü bulmak logaritmik zaman alır.

Aşağıdaki kodda q öncelikli kuyruk $(-d, x)$ şeklinde çiftleri barındırır. Burada çift. x düğüme olan şu anki mesafesinin d olduğu anlamına gelir. Burada mesafeyi negatif tutmamızın sebebi ise spesifik olarak belirtilmediği sürece öncelikli kuyruğun normalde daha büyük sayılara öncelik vermesinden dolayıdır. **distance** dizisi ise her düğüme olan mesafeyi içerir ve **processed** dizisi de bir düğümün işlenip işlenmediğini verir. Başta x düğüme olan uzaklık 0 olup diğer düğümlere olan uzaklık ise ∞ olur.

```
for (int i = 1; i <= n; i++) distance[i] = INF;
distance[x] = 0;
q.push({0,x});
while (!q.empty()) {
    int a = q.top().second; q.pop();
    if (processed[a]) continue;
    processed[a] = true;
    for (auto u : adj[a]) {
        int b = u.first, w = u.second;
        if (distance[a]+w < distance[b]) {
            distance[b] = distance[a]+w;
            q.push({-distance[b],b});
        }
    }
}
```

}

Burada öncelikli kuyruğun *negatif* mesafe tutulduğuna dikkat edilmesi gerekir. Bunun sebebi ise C++'ın varsayılan öncelikli kuyruk yapısında yapı maksimum elemanları başta tutar ama biz minimum elemanların başta olmasını isteriz. Negatif değerleri kullanarak direk olarak varsayılan öncelikli kuyruğu kullanabiliriz³. Bu arada öncelikli kuyrukta aynı düğüme ait birkaç tane eleman olabilir fakat biz en kısa mesafeye sahip olan elemanı işleriz.

Yukarıdaki kodun zaman karmaşıklığı $O(n + m \log m)$ olur çünkü algoritma bütün düğümlerden geçip her kenar için öncelikli kuyruğa en fazla bir eleman eklenir.

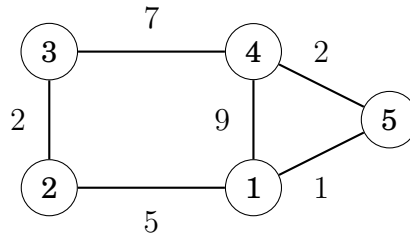
13.3 Floyd–Warshall Algoritması

Floyd–Warshall algoritması⁴ en kısa yolları bulmak için probleme farklı bir çözüm yolu sağlar. Şu ana kadar bu bölümde işlediğimiz diğer algoritmaların aksine Floyd–Warshall algoritması bütün düğümlerden bütün düğümlere olan en kısa yolları tek koşuda bulur.

Algoritma iki boyutlu bir dizi tutar. Bu dizide düğümler arası mesafeler tutulur. İlk başta mesafeler sadece direk düğümler arasındaki kenarlarla hesaplanır. Sonrasında algoritma mesafeleri yollar arasında bulunan ara düğümlerle kısaltmaya çalışır.

Örnek

Floyd–Warshall algoritmasının aşağıdaki çizgede nasıl çalıştığına bakalım:



Başta her düğümün kendisine olan mesafesi 0 olur. Eğer a ve b düğümleri arasında x ağırlığında bir kenar varsa a ve b düğümleri arası mesafe x olur. Diğer bütün mesafeler sonsuzdur.

Bu çizgede baştaki dizi şeklindeki gibidir:

³Tabii Bölüm 4.5'te de yapıldığı gibi kendi öncelikli kuyruğumuzu tanımlayıp pozitif değerleri kullanabiliriz fakat implementasyonu biraz daha uzun olur.

⁴Algoritma R. W. Floyd and S. Warshall kişilerinden adını almıştır. İkisi de algoritmayı birbirinden habersiz bir şekilde 1962'de yayınlamıştır. [23, 70].

	1	2	3	4	5
1	0	5	∞	9	1
2	5	0	2	∞	∞
3	∞	2	0	7	∞
4	9	∞	7	0	2
5	1	∞	∞	2	0

Algoritma ardışık turlardan oluşur. Her turda algoritma yeni bir düğüm seçer ve bu düğüm yollarda ara düğümmüş gibi çalışır. Algoritma, mesafeleri bu düğüme göre azaltmaya çalışır.

İlk tura, 1. düğüm yeni ara düğümdür. 2 ve 4. düğüm arasında 14 mesafeli yeni bir düğüm vardır çünkü 1. düğüm ikisini de bağlar. Bununla beraber 2 ve 5. düğümler arasında 6 uzunluğunda yeni bir yol vardır.

	1	2	3	4	5
1	0	5	∞	9	1
2	5	0	2	14	6
3	∞	2	0	7	∞
4	9	14	7	0	2
5	1	6	∞	2	0

İkinci turda 2. düğüm yeni ara düğüm olur. Böylece 1 ve 3. düğümler arasında ve 3 ve 5. düğümler arasında yeni yollar oluşturur:

	1	2	3	4	5
1	0	5	7	9	1
2	5	0	2	14	6
3	7	2	0	7	8
4	9	14	7	0	2
5	1	6	8	2	0

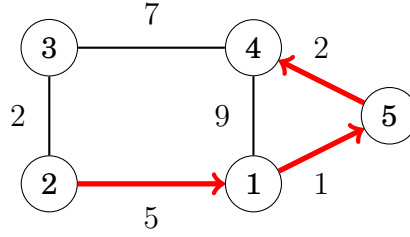
3. turdu, 3. düğüm yeni ara düğüm olur. 2 ve 4. düğüm arasında yeni bir yol oluşur:

	1	2	3	4	5
1	0	5	7	9	1
2	5	0	2	9	6
3	7	2	0	7	8
4	9	9	7	0	2
5	1	6	8	2	0

Algoritma ara düğüm olmamış düğüm kalmayana kadar devam eder. Algoritma bittikten sonra dizi her iki düğüm arasında minimum mesafeleri tutar:

	1	2	3	4	5
1	0	5	7	3	1
2	5	0	2	8	6
3	7	2	0	7	8
4	3	8	7	0	2
5	1	6	8	2	0

Örneğin algoritma bize 2 ve 4. düğümler arasında en kısa yolun 8 mesafesinde olduğunu söyler. Bu aşağıdaki yola karşılık gelir:



Implementasyon

Floyd-Warshall algoritmasının avantajı kodu kolay yazılabiliyor olmasıdır. Aşağıdaki kod bir mesafe matrisi tutar ve $distance[a][b]$ ise a ve b düğümleri arasındaki en kısa mesafeyi verir. Algoritma ilk başta adj komşuluk matrisini kullanarak $distance$ 'ı başlatır:

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        if (i == j) distance[i][j] = 0;
        else if (adj[i][j]) distance[i][j] = adj[i][j];
        else distance[i][j] = INF;
    }
}
```

Bundan sonra en kısa mesafeler aşağıdaki şekilde bulunabilir:

```
for (int k = 1; k <= n; k++) {
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            distance[i][j] = min(distance[i][j],
                                   distance[i][k] + distance[k][j]);
        }
    }
}
```

Algoritmanın zaman karmaşıklığı $O(n^3)$ olur çünkü algoritma çizgedeki her düğümden geçen üç tane iç içe döngü bulundurur.

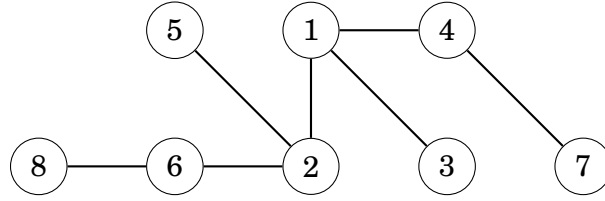
Floyd-Warshall algoritmasının yazılması kolay olduğu için algoritma tek bir kısa mesafe bulmak gerekse bile kullanılabilir. Fakat algoritma çizge eğer kübik zamanın yeterli olabileceği küçüklükteyse kullanılabilir.

Bölüm 14

Ağaç Algoritmaları

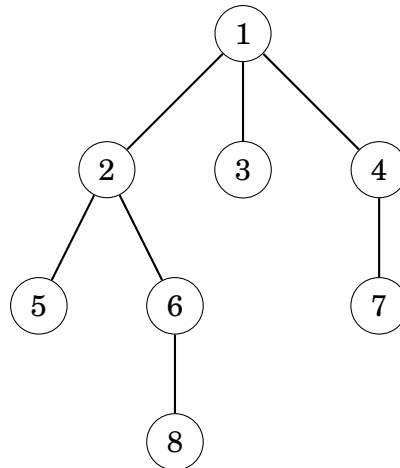
Bir **ağaç**, bağlı n düğüm vwe $n - 1$ kenardan oluşan asiklik (döngü içermeyen) çizgedir. Ağaçtan herhangi bir kenarı çıkartmak ağacı iki parçaya böler ve ağaca herhangi bir kenar eklemek ağaçta döngü oluşturur. Bunlarla beraber ağaçta her iki düğüm arasında tek bir yol vardır.

Örneğin aşağıdaki ağaç 8 düğüm ve 7 kenardan oluşur:



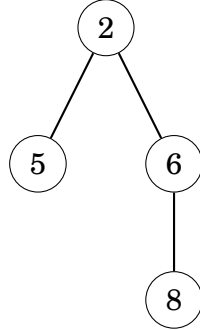
Ağacın **yaprakları** derece sayısı 1 olan yani tek komşuya sahip düğümlerdir. Örneğin yukarıdaki ağacın yaprakları 3, 5, 7 ve 8. düğümlerdir.

Köklü bir ağaçta, düğümlerden biri ağacın **kökü** olur ve diğer bütün düğümler kökün altına yerleştirilir. Örneğin aşağıdaki ağaçta 1. düğüm köktür:



Köklü bir ağaçta bir düğümün **çocukları**, o düğümün alt komşularıdır ve aynı şekilde bir düğümün **ebeveyni** o düğümün üst komşusudur. Her düğümün kök harici bir tane ebeveyni vardır. Kökün ebeveyni yoktur. Örneğin yukarıdaki ağaçta, 2. düğümün çocukları 5 ve 6. düğümken ebeveyni 1. düğümdür.

Köklü bir ağacın yapısı *özyinelemelidir*: Ağaçtaki her düğüm, kendi ve çocuklarının alt ağaçlarında olan düğümleri içeren **alt ağacın** kökü gibi davranır. Örneğin yukarıdaki ağaçta 2. düğümün alt ağacı 2, 5, 6 ve 8. düğümlerden oluşur:



14.1 Tree Dolaşımı

Genel çizge dolaşım algoritmaları ağacın düğümlerini dolaşırken de kullanılabilir. Fakat, ağacın dolaşımını koda geçirmek genel bir çizgeninkini geçirmekten daha kolaydır çünkü ağaç hiç düğüm içermez ve bir düğüme birkaç yoldan ulaşamaz.

Bir ağacı dolaşmanın klasik bir yolu herhangi bir düğümden derinlik öncelikli arama başlatmaktır. Aşağıdaki özyinelemeli fonksiyon bu iş için kullanılabilir:

```
void dfs(int s, int e) {  
    // s düğümü ile bir şeyler yap  
    for (auto u : adj[s]) {  
        if (u != e) dfs(u, s);  
    }  
}
```

Fonksiyona iki parametre verilir: şu anki s düğümü ve bu düğümün ebeveyni olan e düğümü. e düğümünü fonksiyona vermemizin amacı, aramanın sadece ziyaret edilmemiş düğümlere gittiğinden emin olmaktır. Yoksa ebeveyn düğüme geri dönüp sonsuz döngüye girebilir.

Yukarıdaki fonksiyon aramaya x düğümünde başlar:

```
dfs(x, 0);
```

İlk çağrıda $e = 0$ olur çünkü bundan önce başka düğüme gidilmemiştir ve bu yüzden ağaçta istenildiği yönden gidilebilir.

Dinamik Programlama

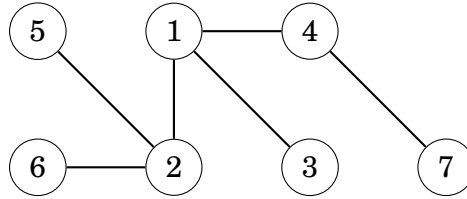
Ağaç dolaşımı yaparken bazı bilgileri hesaplamak için dinamik programlama kullanılabilir. Örneğin dinamik programlama kullanarak $O(n)$ zamanda, ağaçtaki her düğüm için alt ağacında bulunan düğüm sayısını veya bir düğümden yaprağa olan en uzun uzaklık bulunabilir.

Örneğin her s düğümü için alt ağacında toplam düğüm sayısını veren $\text{count}[s]$: değerini bulalım: Alt ağaç, kendisini ve çocuklarının alt ağacındaki bütün düğümleri içerir. Toplam düğüm sayısını özyinelemeli şekilde aşağıdaki kod ile yapabiliriz:

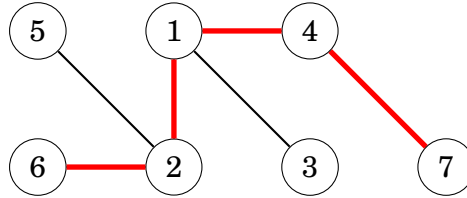
```
void dfs(int s, int e) {
    count[s] = 1;
    for (auto u : adj[s]) {
        if (u == e) continue;
        dfs(u, s);
        count[s] += count[u];
    }
}
```

14.2 Çap

Bir ağacın **çapı** o ağaçtaki iki düğüm arasındaki en uzun uzaklıktır. Örneğin aşağıdaki ağaca baktığımızda:



Bu ağacın çevresi 4'tür ve aşağıdaki yola karşılık gelir:



Bu arada birden fazla en uzun uzaklığa sahip yol olabilir. Yukarıdaki yolda 6. düğümü 5. ile değiştirip 4 uzunluğuna sahip başka bir yol elde edebiliriz.

Şimdi ağaç çapını bulan iki tane $O(n)$ algoritmasından bahsedeceğiz. İlk algoritma dinamik programlama ile yapılır ve ikincisi ise iki tane derinlik öncelikli arama ile yapılır.

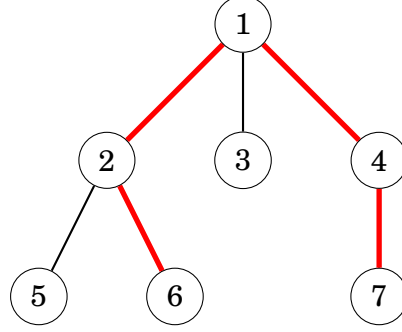
1. Algoritma

Ağaç problemlerine yaklaşmanın genel bir yolu ağacı herhangi bir düğümü kök düğüm yapmaktır. Bundan sonra soruyu her alt ağaç için ayrı ayrı çözmeye çalışabiliriz. İlk algoritmamız çapı hesaplamak için bu fikri kullanmaktadır:

Burada yapılması önemli bir gözlem, köklü bir ağaçtaki her yolun bir *en yüksek* noktası olmasıdır yani yolda geçtiğimiz en yukarıda bulunan düğüm. Böylece

her düğüm için o düğümü en yüksek nokta olarak o düğümünden geçen yolları bulabiliriz:

Örneğin aşağıdaki ağaçta, 1. düğüm aşağıdaki çapta bulunan en yüksek düğündür:



Her x düğümü için iki değer hesaplarız:

- $\text{toLeaf}(x)$: x düğümünden herhangi bir yaprağa olan en uzun n mesafe.
- $\text{maxLength}(x)$: tepe noktası x olan en uzun yol uzunluğu

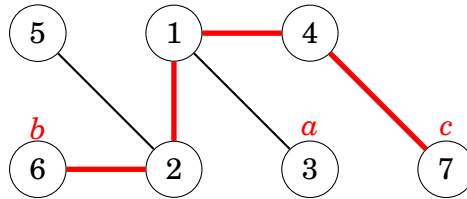
Örneğin yukarıdaki ağaçta $\text{toLeaf}(1) = 2$ olur çünkü $1 \rightarrow 2 \rightarrow 6$ yolu vardır ve $\text{maxLength}(1) = 4$ çünkü $6 \rightarrow 2 \rightarrow 1 \rightarrow 4 \rightarrow 7$ yolu vardır. Bu durumda $\text{maxLength}(1)$ çapa eşittir.

Dinamik programlama her düğüm için yukarıdaki değerleri $O(n)$ zamanda hesaplayabilir. İlk başta $\text{toLeaf}(x)$ değerini hesaplamak için x düğümünün çocuklarını ziyaret ederiz ve $\text{toLeaf}(c)$ değeri maksimum olan bir c çocuğu seçip bu değeri bir arttırırız. Sonra $\text{maxLength}(x)$ değerini hesaplamak için iki farklı a ve b çocuk düğümü seçeriz ki bunların $\text{toLeaf}(a) + \text{toLeaf}(b)$ maksimum olur ve bu toplama iki ekleriz.

Algorithm 2

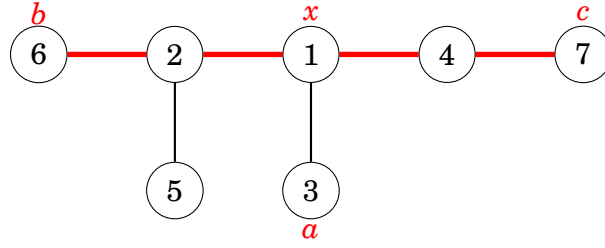
Çapı hesaplamamanın verimli yollarından biri de iki tane derinlik öncelikli arama yapmaktır. İlk başta ağaçta bir a düğümü seçeriz ve a 'dan en uzakta olan bir b düğümü buluruz. Sonrasında b düğümünden en uzakta olan bir c düğümü buluruz. Ağacın çapı b ve c arasındaki mesafedir.

Aşağıdaki çizgede a , b , c :



olabilir. Bu çok sık bir çözümdür ama neden doğru çalışır?

Bunu görmek için ağactaki çapı yatay çizecek şekilde bir kere daha çizmeliyiz ve geri kalan düğümleri bu yatay yoldan sarkacak şekilde çizmeliyiz:

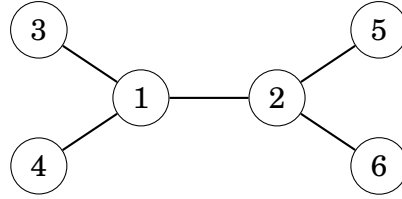


x düğümüm a düğümünden gelen yolun çapa katıldığı yeri gösterir. a düğümünden en uzak düğüm b , c veya diğerleri kadar en az x düğümünden uzak olacak kadar uzakta olan bir düğüm olur. Böylece bu düğüm her zaman çapa karşılık gelen yolun bitiş veya başlangıç düğümü olur.

14.3 Bütün En Uzun Yollar (All Longest Paths)

Şimdi, bizden her nokta için o noktadan başlayan en uzun yolu bulmamız isteniyor: Bu ağaç çapı probleminin genel hale geçirilmiş sorusu gibi görünebilir çünkü bu yolların en uzağı ağacın çapına eşittir. Bu problem $O(n)$ zamanda çözülebilir:

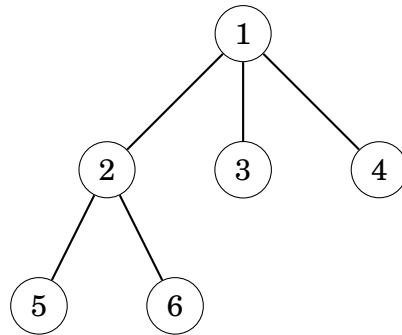
Örneğin aşağıdaki ağacı düşünelim:



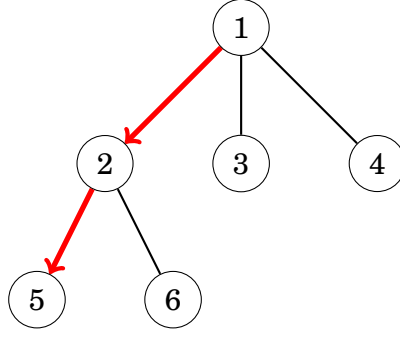
$\text{maxLength}(x)$, x düğümünden başlayan en uzun yolun uzunluğunu verir. Örneğin yukarıdaki ağaçta $\text{maxLength}(4) = 3$ olur çünkü $4 \rightarrow 1 \rightarrow 2 \rightarrow 6$ yolu vardır. Aşağıdaki tablo bütün değerleri verir:

düğüm x	1	2	3	4	5	6
$\text{maxLength}(x)$	2	2	3	3	3	3

Bu probleme başlamak için ağaçta herhangi bir düğümü kök düğüm yapabiliriz:

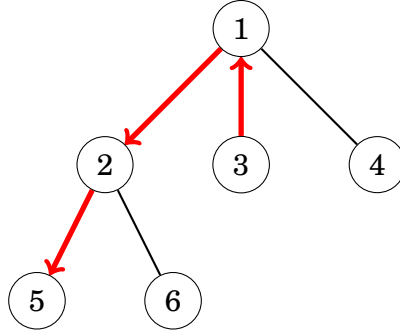


Problemin ilk kısmında, her x düğümü için x düğümünün bir çocuğundan geçen en uzun yolu hesaplamak olacak. Örneğin 1. düğümünden çocuğu olan 2. düğümünden geçen yol en uzundur:

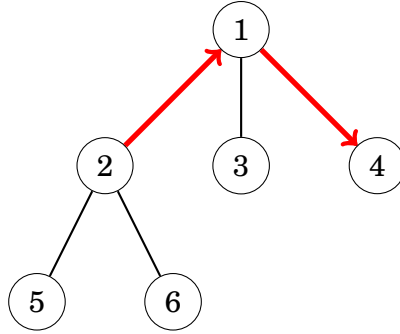


Bu problem öncesinde de yaptığımız gibi dinamik programlama kullanarak $O(n)$ zamanda çözebiliriz.

Sonra sorunun 2. kısmında, her x düğümü için ebeveyni p düğümünden geçen en uzun yolun uzunluğunu bulmalıyız. Örneğin 3. düğüm için ebeveyni 1. düğümünden geçen en uzun yol aşağıdaki gibidir:



İlk bakışta, p düğümünden en uzun yolu seçmemiz gerekiyormuşuz gibi görünür fakat bu her zaman doğru *değildir* çünkü p düğümünden geçen en uzun yol x düğümünden geçebilir. Bu durumun örneği aşağıdaki gibidir:



Yine de 2. kısmı her x düğümü için *iki* tane maksimum uzunluğu kaydederek $O(n)$ zamanda çözebiliriz:

- $\text{maxLength}_1(x)$: x düğümünden geçen en uzun yolun uzunluğu
- $\text{maxLength}_2(x)$ ilk yoldan farklı bir yönde olan x düğümünden geçen en uzun yolun uzunluğu

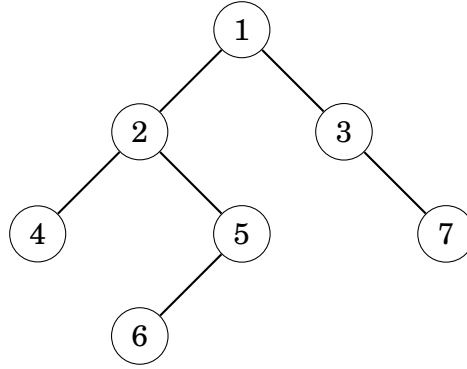
Örneğin yukarıdaki çizgede $1 \rightarrow 2 \rightarrow 5$ yolunu kullanarak $\text{maxLength}_1(1) = 2$ ve $1 \rightarrow 3$ yolunu kullanarak $\text{maxLength}_2(1) = 1$ olur.

En sonunda eğer $\text{maxLength}_1(p)$ değerini sağlayan yol x düğümünden geçiyorsa, maksimum uzunluğun $\text{maxLength}_2(p) + 1$ olduğunu ve eğer geçmiyorsa da $\text{maxLength}_1(p) + 1$ olduğunu anlarız.

14.4 İkili Ağaç (Binary Tree)

İkili ağaç, her düğümün sol ve sağ diye iki tane alt ağacı bulunan köklü bir ağaçtır. Bir düğümün alt ağacının boş olabilmesi muhtemeldir. Böylece ikili ağacının her düğümünün 0, 1 veya 2 çocuğu olabilir.

Örneğin aşağıdaki ağaç bir ikili ağaçtır:



İkili ağacın düğümlerinin ağacı dolaşma şekline göre 3 farklı sıralanması vardır:

- **önce kök (pre-order)**: İlk kökü işler sonra sol alt ağacı dolaşır ve en son sağ alt ağacı dolaşır. then traverse the left subtree, then traverse the right subtree
- **kök ortada (in-order)**: İlk sol alt ağacı dolaşır sonra kökü işler ve en son sağ alt ağacı dolaşır.
- **kök sonda (post-order)**: ilk sol alt ağacı sonra sağ alt ağacı dolaşır ve en son kökü işler.

Yukardaki ağaçta önce köke göre sıralama $[1, 2, 4, 5, 6, 3, 7]$, kök ortadaya göre $[4, 2, 6, 5, 1, 3, 7]$ ve kök sondaya göre $[4, 6, 5, 2, 7, 3, 1]$ şeklinde olur.

Eğer ağacın önce kök ve kök ortada sırasını biliyorsak, ağacın tam yapısını oluşturabiliriz. Örneğin yukarıdaki ağaç, $[1, 2, 4, 5, 6, 3, 7]$ önce kök sırasına ve $[4, 2, 6, 5, 1, 3, 7]$ kök ortada sırasına sahip olabilecek tek ağaçtır. Benzer şekilde kök sonda ve kök ortada sıralamaları da ağacın yapısını belirleyebilir.

Fakat bir ağacın sadece önce kök ve kök sonda sıralamalarını bildiğimiz zaman durum değişir. Bu durumda birden çok ağaç bu sıralamalara uyabilir. Örneğin aşağıdaki iki ağaçta da



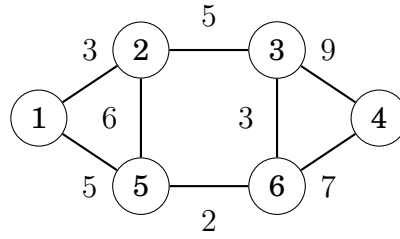
önce kök sıralaması $[1,2]$ ve kök sonda sıralaması $[2,1]$ olur ama bu ağaçların yapıları birbirlerinden farklıdır.

Bölüm 15

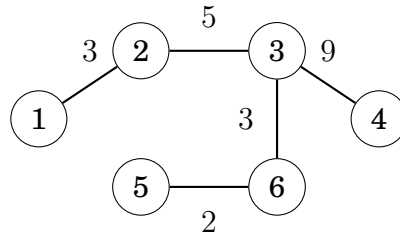
Kapsayan Ağaç (Spanning Trees)

Kapsayan ağaç bir çizgenin bütün düğümlerini bağlı olacak şekilde birleştiren çizgenin bazı kenarlarını içeren bir ağaçtır. Ağaçlardaki gibi, kapsayan ağaçlar da bağlı ve asiklidir. Genelde, kapsayan ağaç oluşturma'nın birkaç yolu vardır.

Örneğin aşağıdaki çizgeye bakalım:

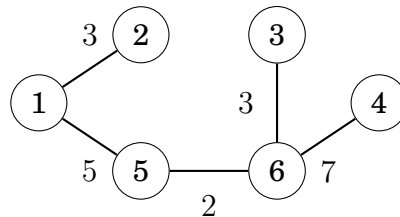


Bir tane kapsayan ağaç aşağıdaki gibidir:

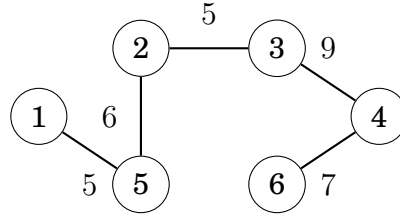


Kapsayan ağacın ağırlığı kenar ağırlıklarının toplamıdır. Örneğin yukarıdaki kapsayan ağacın ağırlığı $3 + 5 + 9 + 3 + 2 = 22$ olur.

En küçük kapsayan ağaç (minimum spanning tree), ağırlığı en küçük olan kapsayan ağaçtır. Yukarıdaki çizgenin, en küçük kapsayan ağacının ağırlığı 20'dir ve aşağıdaki gibi oluşturulabilir:



En büyük kapsayan ağaç, benzer şekilde en büyük ağırlığa sahip kapsayan ağaçtır. Yukarıdaki çizgenin en büyük kapsayan ağaç 32'dir:



Bir çizgenin birkaç tane en küçük ve en büyük kapsayan ağacı olabilir yani bu ağaçlardan sadece bir tane olmak zorunda değildir.

Bazı açgözlü methodları kullanarak en küçük ve en büyük kapsayan ağaçları oluşturabiliriz. Bu bölümde kenarların ağırlıklarına göre sıralayarak yapılan iki tane algoritmadan bahsedeceğiz. Her ne kadar bölümde en küçük kapsayan ağaçları bulmaya odaklanacak olsak bile en büyük kapsayan ağaç da kenarları ters sırada işleyerek bulabiliriz.

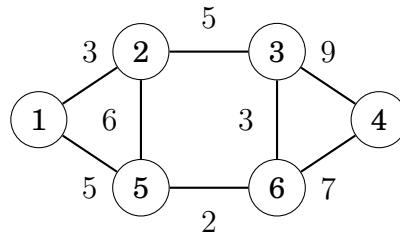
15.1 Kruskal'ın Algoritması

Kruskal'ın Algoritması'nda¹, baştaki kapsayan ağaçta sadece çizgenin düğümleri bulunur ve herhangi bir kenar içermemektedir. Sonrasında algoritma ağırlıklarına göre kenarlara bakar ve eğer kenar döngü oluşturmuyorsa kenarı kapsayan ağaca ekler.

Algoritma, ağacın parçalarını tutar. İlk başta çizgenin her düğümü ayrı bir parçadır. Her seferinde ağaca bir kenar eklendiği zaman iki parça birleşir. Sonda bütün düğümler aynı parçaya ait olur ve en küçük kapsayan ağaç bulunur.

Örnek

Kruskal'ın Algoritması'nın aşağıdaki çizgeyi nasıl işlediğine bakalım:



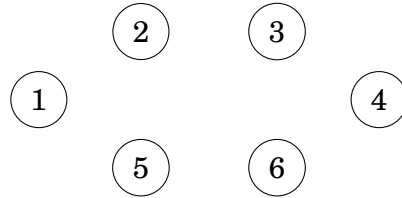
Algoritmasının ilk adımında kenarlar, ağırlıklarına göre küçükten büyüğe doğru sıralanır. Sonuç olarak gelecek sıra:

¹Algoritma 1956 yılında J. B. Kruskal tarafından yayınlanmıştır [48].

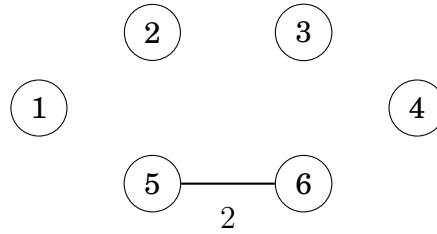
kenar	ağırlık
5-6	2
1-2	3
3-6	3
1-5	5
2-3	5
2-5	6
4-6	7
3-4	9

Bundan sonra, algoritma listeden geçer ve kenar iki ayrı parçayı bağlıyorsa kenarı ağaca ekler.

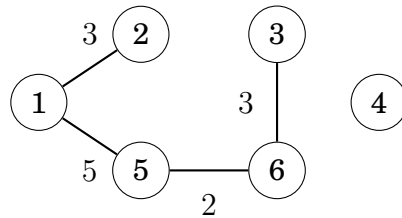
Başta her düğüm kendisine ait parçadadır:



Ağaca ilk eklenecek kenar 5-6 kenarıdır ve bu kenar {5} ve {6} parçalarını {5,6} şeklinde birleştirir:



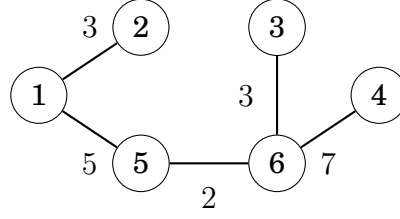
Bundan sonra 1-2, 3-6 ve 1-5 kenarları benzer şekilde eklenir:



Bu adımlardan sonra, çoğu parçalar birleşmiştir ve sonuç olarak ağaçta iki parça vardır: {1,2,3,5,6} ve {4}.

Listedeki sonraki kenar 2-3 olur ama bu kenar ağaca eklenmez çünkü 2 ve 3. düğümler aynı parçaya aittir. Aynı nedenden dolayı 2-5 kenarı da ağaca eklenmez.

En sonunda 4-6 kenarı ağaca eklenir:

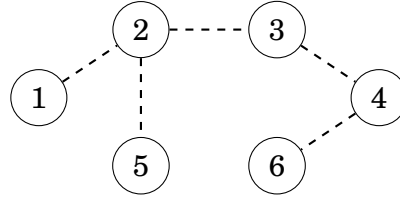


Bundan sonra algoritma herhangi bir kenar eklenmez çünkü ağaç bağlanmıştır ve her iki düğüm arasında bir yol vardır. Sonuç olarak oluşan en küçük kapsayan ağacın ağırlığı $2 + 3 + 3 + 3 + 5 + 7 = 20$ olur.

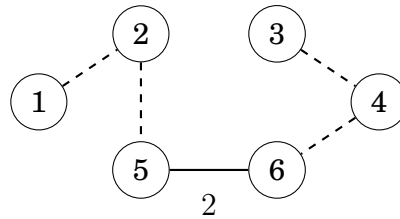
Bu Neden Çalışır?

Kruskal'ın Algoritması'nın neden çalıştığı güzel bir sorudur. Neden açgözlü bir strateji bize en küçük kapsayan ağacı bulmayı garantiler?

Çizgedeki minimum ağırlıklı kenarı **eklemezsek** ne olacağına bakalım. Örneğin yukarıdaki çizgenin 5-6 kenarı içermediğini varsayalım. Böyle bir kapsayan ağacın tam yapısını bilmiyoruz ama yine de her durumda bazı kenarları içerir. Varsayalım ki ağaç aşağıdaki gibi olsun:



Fakat, yukarıdaki ağacın en küçük kapsayan ağaç mümkün değildir çünkü bu ağaçtan bir kenarı çıkartıp yerine 5-6 kenarını eklersek daha *küçük* ağırlığa sahip bir kapsayan ağaç oluşur:



Bu nedenden dolayı en küçük ağırlığa sahip kenarı eklemek bize her zaman en küçük kapsayan ağacı oluşturmayı sağlar. Benzer mantığı kullanarak sonraki kenarları da ağırlık sıralamasına göre almanın mantıklı olacağını gösterebiliriz. Bundan dolayı Kruskal'ın Algoritması doğru çalışır ve her zaman en küçük kapsayan ağacı oluşturur.

Implementasyon

Kruskal'ın Algoritmasını koda geçirirken, çizgeyi bir kenar listesinde tutmak daha rahat olur. Algoritmanın ilk aşamasında listedeki kenarlar $O(m \log m)$ zamanda sıralanır. Bundan sonra algoritmanın ikinci aşamasında algoritma en küçük kapsayan ağacı şeklindeki gibi oluşturur:

```
for (...) {  
    if (!same(a,b)) unite(a,b);  
}
```

Döngü listedeki bütün kenarlardan geçer ve her zaman a b düğümleri arasındaki a – b kenarlarını işler. İki fonksiyon gereklidir: `same` fonksiyonu a ve b düğümlerinin aynı parçaya ait olup olmadığını belirler ve `unite` fonksiyonu da a ve b düğümlerini içeren parçaları birleştirir.

Buradaki sorun `same` ve `unite` fonksiyonlarını verimli bir şekilde implement etmektir. Yapılabilecek şeylerden biri `same` fonksiyonunu bir çizge dolaşım algoritması yapıp a düğümünden b düğüme gidilebiliyor mu diye kontrol etmektir. Fakat bu durumda bu fonksiyonun zaman karmaşıklığı $O(n+m)$ olur ve sonuç olarak yapılacak algoritma yavaş olur çünkü `same` fonksiyonu çizgenin her kenarında çağrılır.

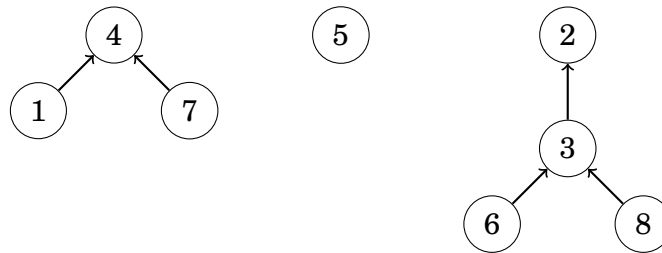
Bu problemi union-find yapısı ile iki fonksiyonu da $O(\log n)$ zamanda çalıştırabileceğiz. Böylece Kruskal'ın Algoritması listeyi sıraladıktan sonra $O(m \log n)$ zamanda çalışır.

15.2 Union-find Yapısı

union-find yapısı parçalardan oluşan bir koleksiyonu tutar. Her eleman sadece bir parçaya aittir yani bu parçalar ayrıklardır. İki tane $O(\log n)$ işlemi vardır: `unite` işlemi iki parçayı birleştirir ve `find` işlemi, verilen elemanın bulunduğu parçayı verir².

Yapı

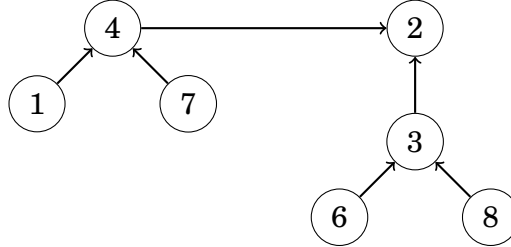
Union-find yapıda, her parçadaki bir eleman o parçanın temsilcisi olur ve parçadaki diğer bütün elemanlardan temsilciye bir bağ vardır. Örneğin $\{1,4,7\}$, $\{5\}$ ve $\{2,3,6,8\}$ diye parçalarımızın olduğunu düşünelim:



²Burada verilen yapı 1971 yılında J. D. Hopcroft ve J. D. Ullman tarafından tanıtılmıştır [38]. Sonra 1975 yılında R. E. Tarjan, bu yapının günümüzde çoğu algoritma kitabında bahsedilen daha gelişmiş bir versiyonunu bulmuştur [64].

Bu durumda bu parçaların temsilcileri 4, 5 ve 2 olur. Her elemanın temsilcisini bu elemanda başlayan bağı takip ederek bulabiliriz. Örneğin 2. eleman 6. elemanın temsilcisidir çünkü $6 \rightarrow 3 \rightarrow 2$ diye bir bağ var. İki elemanın temsilcisi aynı ise bu elemanlar aynı parçada bulunmaktadır.

İki parça, parçaların temsilcilerini bağlayarak birleştirilebilir. Örneğin $\{1, 4, 7\}$ ve $\{2, 3, 6, 8\}$ parçaları aşağıdaki gibi birleştirilebilir:



Sonuç olarak oluşacak parça $\{1, 2, 3, 4, 6, 7, 8\}$ elemanlarını içerir. Bundan sonra 2. eleman bütün parçanın temsilcisi olur ve artık eski temsilci 4. eleman, 2. elemanı temsilci olarak gösterir.

Union-find yapısının verimliliği parçaların nasıl bağlandığına bağlıdır. Basit bir strateji kullanabiliriz: Her zaman *küçük* olan parçanın temsilcisini *büyük* olan parçanın temsilcisine (ve eğer iki parça da aynı büyüklükteyse aralarından birini) bağlarız. Bu tekniği kullanarak herhangi bir bağı uzunluğu $O(\log n)$ olur ve herhangi bir elemanın temsilcisini onun bağına göre verimli bir şekilde bulabiliriz.

Implementasyon

Union-find yapısı dizileri kullanarak koda geçirilebilir. Aşağıdaki kodda `link` dizisi her elemanın bağıdaki sonraki elemanı veya eleman temsilci ise kendisini tutar. `size` dizisi ise her temsilcinin bulunduğu parçaların büyüklüğünü verir.

Başta, her eleman ayrı bir parçaya aittir:

```
for (int i = 1; i <= n; i++) link[i] = i;
for (int i = 1; i <= n; i++) size[i] = 1;
```

`find` fonksiyonu herhangi bir x elemanının temsilcisini bulur. Temsilci x elemanından başlayan düğümü kullanarak aşağıdaki gibi bulunabilir:

```
int find(int x) {
    while (x != link[x]) x = link[x];
    return x;
}
```

`same` fonksiyonu ise a ve b düğümlerinin aynı parçaya ait olup olmadığını verir. Bu kolay bir şekilde `find` fonksiyonu ile yapılabilir:

```
bool same(int a, int b) {
    return find(a) == find(b);
}
```

`unite` fonksiyonu ise a ve b elemanlarını içeren parçaları birleştirir (elemanların farklı parçalarda olması gerekir.). Fonksiyon ilk başta parçaların temsilcilerini bulur ve ondan sonra küçük parçayı büyük parçaya bağlar:

```
void unite(int a, int b) {
    a = find(a);
    b = find(b);
    if (size[a] < size[b]) swap(a,b);
    size[a] += size[b];
    link[b] = a;
}
```

`find` fonksiyonunun zaman karmaşıklığı her bağın uzunluğunun $O(\log n)$ varsayarak $O(\log n)$ zamanda çalışır. Bu durumda `same` ve `unite` fonksiyonları da $O(\log n)$ zamanda çalışır. `unite` fonksiyonu küçük olan parçayı büyük olan parçaya bağlayarak her bağın $O(\log n)$ uzunluğunda olduğunu sağlar.

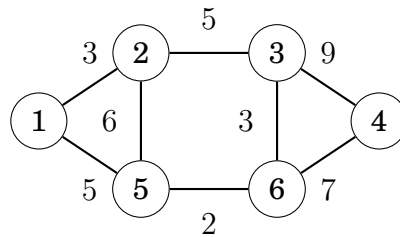
15.3 Prim'in Algoritması

Prim'in Algoritması³ en küçük kapsayan ağacı bulmak konusunda alternatif bir yöntemdir. Bu algoritma ilk başta çizgeden herhangi bir düğümü ağaca ekler. Sonra, algoritma her zaman ağaca yeni bir düğüm ekleyen en kısa kenarı ekler. En sonda bütün düğümler ağaca eklenir ve en küçük kapsayan ağaç bulunmuş olur.

Prim'in Algoritması Dijkstra'nın Algoritmasını anımsatır. Aralarındaki fark, Dijkstra'nın Algoritması, her zaman başlangıç düğümünden en kısa mesafede olan kenarı seçerken Prim'in algoritması ağaca yeni bir düğüm eklendikten sonra en kısa kenarı seçer.

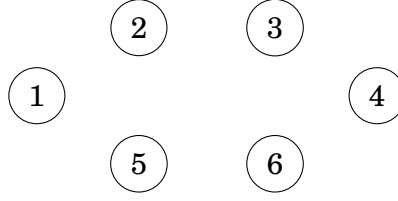
Örnek

Aşağıdaki çizgede Prim'in Algoritmasının nasıl çalıştığına bakalım:

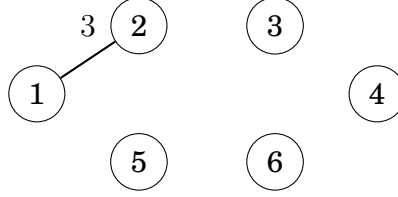


İlk başta hiçbir düğüm arasında kenar yoktur:

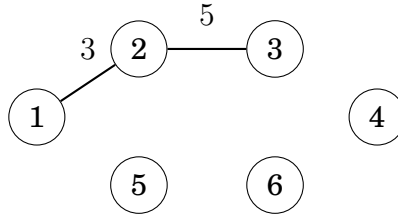
³Algoritma R. C. Prim tarafından ismini almıştır. Prim bu algoritmayı 1957 yılında yayınlamıştır [54]. Fakat aynı algoritma zaten 1930 yılında V. Jarnik tarafından keşfedilmiştir.



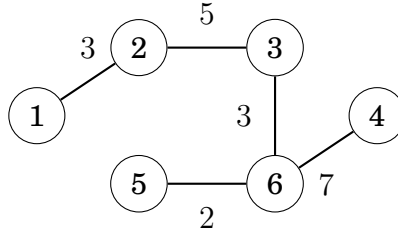
Başlangıç düğümü olarak herhangi bir düğüm seçilir o yüzden 1. düğümü seçelim. İlk başta 3 uzunluğunda olan kenarı eklersek 2. düğümü ekleriz:



Bundan sonra, 5 uzunluğunda olan iki kenar vardır ve böylece 3. düğümü ve 5. düğümü ağaca ekleyebiliriz. İlk 3. düğümü ağaca ekleyelim:



İşlem, bütün düğümler ağaca eklenene kadar devam eder:



Implementasyon

Dijkstra'nın Algoritması gibi Prim'in Algoritmasının da öncelikli kuyruk ile verimli bir şekilde kodu yazılabilir. Öncelikli kuyruk tek kenarı kullanarak ekleyebilen bütün düğümleri tutması gerekmektedir. Bu düğümler kenarların artan ağırlığına göre sıralanır.

Prim'in Algoritmasının zaman karmaşıklığı $O(n + m \log m)$ olup Dijkstra'nın Algoritmasının zaman karmaşıklığıyla aynıdır. Pratikte hem Prim'in hem de Kruskal'ın Algoritmaları verimlidir ve hangi algoritmanın kullanılacağı kişiden kişiye göre değişir. Yine de çoğu olimpiyatçı Kruskal'ın Algoritmasını kullanır.

Bölüm 16

Yönlü Çizgeler

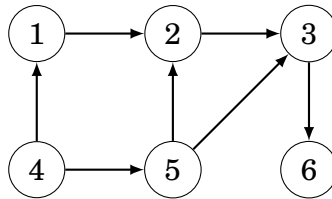
Bu bölümde yönlü çizgelerin iki türünden bahsedeceğiz:

- **Asiklik Çizgeler (Acyclic Graphs):** Çizgede hiçbir döngü yoktur yani bir düğümden kendine gelen bir yol yoktur¹.
- **Varis Çizgeleri (Successor Graphs):** Her düğümden çıkan sadece 1 kenar vardır yani her düğümün farklı bir ardılı vardır.

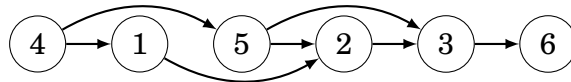
İki durumda da bu özellikleri kullanarak çeşitli verimli algoritmalar tasarlayabiliriz.

16.1 Topolojik Sıralama (Topological Sorting)

Topolojik sıralama yönlü çizgenin düğümlerini sıralama şeklidir. Sıralamanın mantığı ise eğer a düğümünden b düğüme bir yol varsa o zaman a düğümü b düğümünden daha önce sırada bulunur. Örneğin aşağıdaki çizgede



bir tane topolojik sıralama $[4, 1, 5, 2, 3, 6]$:



şeklinde olur. Asiklik çizgelerinde her zaman topolojik sıralama vardır fakat çizge döngü içeriyorsa topolojik sıralama yapmak mümkün değildir çünkü döngüdeki hiçbir düğüm sıralamada diğer düğümlerden önce gelemez. Derinlik öncelikli arama kullanarak yönlü bir çizgenin döngü içerip içermediğini bulabiliriz ve eğer içermiyorsa topolojik sıralama yaparız.

¹Yönlü asiklik çizgeler bazen DAG olarak isimlendirilir.

Algoritma

Buradaki fikir çizgenin bütün düğümlerinden geçip işlenmemiş her düğümden derinlik öncelikli arama başlatmaktır. Aramalarda düğümlerin 3 olası durumu vardır:

- 0. durum: düğüm işlenmedi (beyaz)
- 1. durum: düğüm işleniyor (açık gri)
- 2. durum: düğüm işlendi (koyu gri)

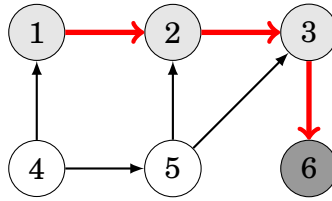
Başta her düğümün durumu 0'dır ve bir düğümden ilk defa arama başladığı zaman düğümün durumu 1 olur. En sonunda düğümün bütün devam düğümleri işlenir ve durumu 2 olur.

Eğer çizge döngü içeriyorsa arama sırasında durumu 1 olan bir düğümlerle karşılaşacağız. Bu durumda topolojik sıralama yapmak mümkün olmayacaktır.

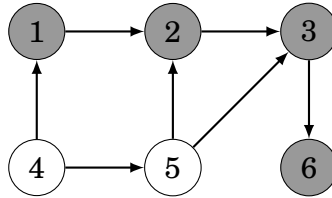
Eğer çizge döngü içermiyorsa düğümü durumu 2 olunca düğümü listeye ekleriz. Topolojik sıra bu listenin tersidir.

1. Örnek

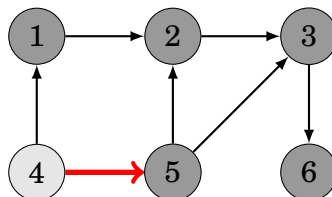
Aşağıdaki örnek çizgede arama ilk 1. düğümden 6. düğüme ilerler:



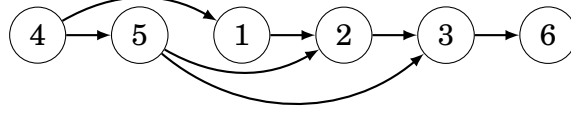
6. düğüm işlenir ve listeye eklenir. Bundan sonra 3, 2. ve 1. düğümler listeye eklenir:



Bu durumda liste [6,3,2,1] olur. Sonraki arama 4. düğümden başlar:



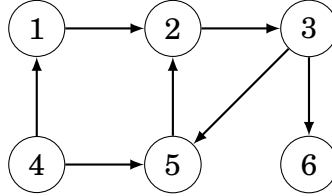
Böylece final liste $[6, 3, 2, 1, 5, 4]$ olur. Bütün düğümleri işledik ve böylece topolojik bir sıralama bulduk. Topolojik sıralama listenin tersi olup $[4, 5, 1, 2, 3, 6]$ olur:



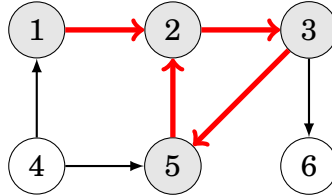
Bir çizgede sadece bir tane topolojik sıralama olmak zorunda değildir ve birkaç tane topolojik sıralama bulunabilir.

2. Örnek

Şimdi döngü içermesinden dolayı topolojik sıralama yapamayacağımız bir çizgeye bakalım:



Arama aşağıdaki gibi ilerler:



Arama durumu 1 olan 3. düğüme varır. Bu çizgede döngü olduğu anlamına gelir. Örnekte $2 \rightarrow 3 \rightarrow 5 \rightarrow 2$ döngüsü vardır.

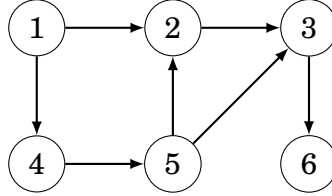
16.2 Dinamik Programlama

Eğer yönlü çizge asiklik ise, o çizgede dinamik programlama yapılabilir. Örneğin aşağıdaki gibi başlangıç düğümünden bitiş düğümüne giden yolları soran problemleri dinamik programlama kullanarak verimli bir şekilde çözebiliriz:

- kaç tane yol var?
- en kısa/uzun yol hangisidir?
- bir yoldaki minimum/maximum kenar sayısı kaçtır?
- hangi düğümler kesinlikle her yolda bulunur?

Yol Sayısını Hesaplama

Örnek olarak aşağıdaki çizgede bulunan 1. düğümden 6. düğüme olan yol sayısını hesaplayalım:



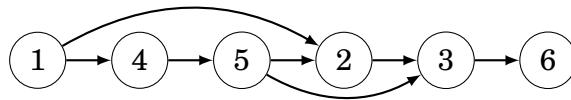
Toplam 3 yol bulunmaktadır:

- $1 \rightarrow 2 \rightarrow 3 \rightarrow 6$
- $1 \rightarrow 4 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 6$
- $1 \rightarrow 4 \rightarrow 5 \rightarrow 3 \rightarrow 6$

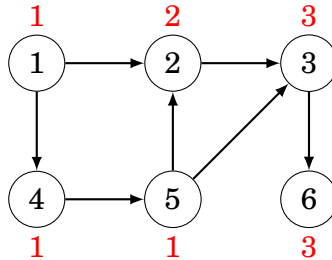
$\text{paths}(x)$ 1. düğümden x düğüme olan yol sayısını verdiğini düşünelim. Temel durumda $\text{paths}(1) = 1$ olur. Sonra diğer $\text{paths}(x)$ değerlerini hesaplamak için

$$\text{paths}(x) = \text{paths}(a_1) + \text{paths}(a_2) + \dots + \text{paths}(a_k)$$

özyinelemesini kullanabiliriz. Burada a_1, a_2, \dots, a_k düğümleri x düğüme giden kenarları bulunan düğümleri belirtmektedir. Çizge asiklik olduğu için $\text{paths}(x)$ değerleri topolojik sırasının sırası ile hesaplanabilir. Yukarıdaki çizgini topolojik sıralaması aşağıdaki gibidir:



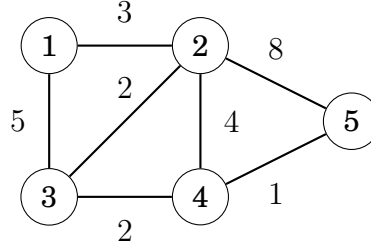
Böylece yol sayıları aşağıdaki gibi olur:



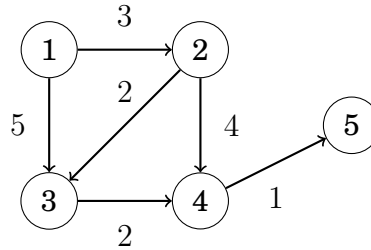
Örneğin $\text{paths}(3)$ değerini hesaplamak için $\text{paths}(2) + \text{paths}(5)$ formülünü kullanabiliriz çünkü 2 ve 5. düğümden 3. düğüme kenar vardır. $\text{paths}(2) = 2$ ve $\text{paths}(5) = 1$ olduğu için $\text{paths}(3) = 3$ olur.

Dijkstra'nın Algoritmasını Genişletmek

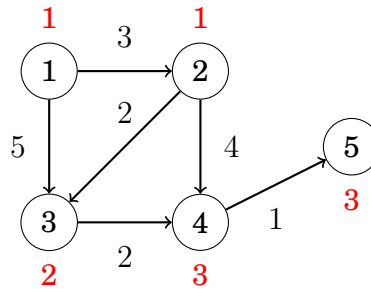
Dijkstra'nın Algoritması'nın yan ürünü olarak yönlü asiklik bir çizge verir. Bu çizge, her düğüm için başlangıç düğümünden başlayarak o düğüme ulaşılacak en kısa yolları gösterir. Dinamik programlama bu çizge üzerinde kullanılabilir. Örneğin



çizgesinde 1. düğümden çıkan en kısa yollar aşağıdaki kenarları kullanabilir:



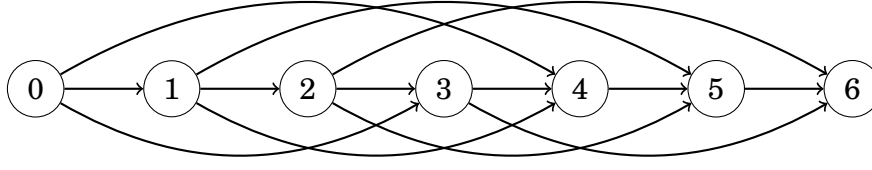
Örneğin şimdi dinamik programlama ile 1. düğümden 5. düğüme en kısa yol sayısını hesaplayabiliriz:



Problemleri Çizge Halinde Göstermek

Aslında herhangi bir dinamik programlama sorusu yönlü asiklik çizge şeklinde gösterilebilir. Böyle bir çizgede her düğüm bir dinamik programlama durumunu gösterir ve kenarlar da bu durumların birbirine nasıl bağlı olduğunu gösterir:

Örneğin $\{c_1, c_2, \dots, c_k\}$ paralarıyla n miktarında para oluşturmamız gerektiğini düşünelim. Bu problemi bir çizge şeklinde gösterebiliriz. Çizgede her düğüm bir para toplamını ve her kenar da paraların nasıl seçilebildiğini gösterir. Örneğin $\{1, 3, 4\}$ ve $n = 6$ için çizge aşağıdaki gibi olur:



Bu gösterimi kullanarak 0. düğümden n . düğüme olan en kısa yol, en az para kullanımına denk gelir. 0. düğümden n . düğüme olan en kısa yol sayısı toplam çözüm sayısını verir.

16.3 Ardıl Yollar (Successor Paths)

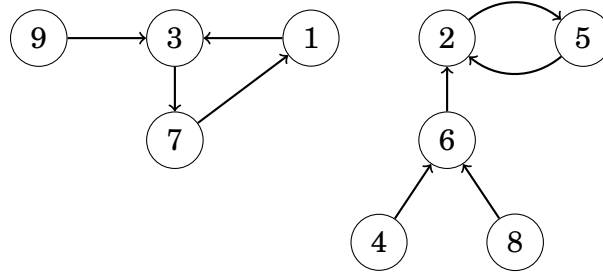
Bu bölümün geri kalanında **ardıl çizgelerden** bahsedeceğiz. Bu çizgelerde her düğümden sadece 1 kenar çıkar. Ardıl çizge bir veya birkaç tane parçadan oluşabilir ve her parça bir döngü ve ona götüren birkaç yol içerir.

Ardıl çizgeler bazen **fonksiyonel çizge** (functional graph) olarak da adlandırılır. Bunun nedeni ise herhangi bir ardıl çizge onun kenarlarını tanımlayan bir fonksiyona karşılık gelir. Fonksiyon parametresi çizgenin bir düğümü olur ve fonksiyon o düğümün ardılıdır.

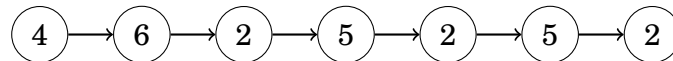
Örneğin

x	1	2	3	4	5	6	7	8	9
$\text{succ}(x)$	3	5	7	6	2	2	1	6	3

fonksiyonu aşağıdaki çizgeye karşılık gelir:



Ardıl çizgede her düğümün tek bir ardılı olduğu için $\text{succ}(x, k)$ diye bir fonksiyon tanımlayabiliriz. Bu fonksiyon x düğümünden başlayarak k adım atarak ulaşabileceğimiz düğümü gösterir. Örneğin yukarıdaki çizgede $\text{succ}(4, 6) = 2$ olur çünkü 4. düğümden 6 adım atarak 2. düğüme varabiliriz.



$\text{succ}(x, k)$ değerini hesaplamamanın düz bir yolu x düğümünden başlayıp k adım atarak vardığımız düğümü bulmaktır. Bu $O(k)$ zaman alır. Fakat, ön işleme (preprocessing) yaparak herhangi bir $\text{succ}(x, k)$ değerini sadece $O(\log k)$ zamanda bulabiliriz.

Buradaki fikir k değeri 2'nin kuvvetleri ve u 'dan (u en fazla yüreğemiz adım sayısıdır) küçük olan bütün $\text{succ}(x, k)$ hesaplamaktır. Bu verimli bir şekilde aşağıdaki özyineleme kullanılarak yapılabilir:

$$\text{succ}(x, k) = \begin{cases} \text{succ}(x) & k = 1 \\ \text{succ}(\text{succ}(x, k/2), k/2) & k > 1 \end{cases}$$

Değerleri önceden hesaplamak $O(n \log u)$ zaman alır çünkü her düğüm için $O(\log u)$ tane değer hesaplanır. Yukarıdaki çizgede ilk değerler aşağıdaki gibidir:

x	1	2	3	4	5	6	7	8	9
$\text{succ}(x, 1)$	3	5	7	6	2	2	1	6	3
$\text{succ}(x, 2)$	7	2	1	2	5	5	3	2	7
$\text{succ}(x, 4)$	3	2	7	2	5	5	1	2	3
$\text{succ}(x, 8)$	7	2	1	2	5	5	3	2	7
...									

Bundan sonra herhangi bir $\text{succ}(x, k)$ değeri k adım sayısını 2'in kuvvetlerinin toplamı şeklinde ifade ederek gösterebiliriz. Örneğin $\text{succ}(x, 11)$ değerini hesaplamak istiyorsak $11 = 8 + 2 + 1$ gösterimini kullanabiliriz. Bunu kullanarak

$$\text{succ}(x, 11) = \text{succ}(\text{succ}(\text{succ}(x, 8), 2), 1).$$

Örneğin önceki çizgede

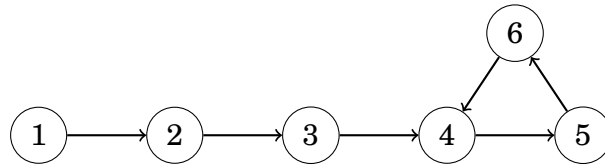
$$\text{succ}(4, 11) = \text{succ}(\text{succ}(\text{succ}(4, 8), 2), 1) = 5.$$

Bu tip bir gösterim her zaman $O(\log k)$ parçadan oluşur yani $\text{succ}(x, k)$ değerini hesaplamak $O(\log k)$ zaman alır.

16.4 Döngü Bulma

Sadece bir tane yolu bulunan bir ardıl çizge düşünelim. Bu yol da döngüyle bitmektedir. Bu durumda şu soruları sorabiliriz: Eğer başlangıç düğümünden ilerlemeye başlarsak döngüdeki ilk düğüm hangisi olur ve döngü kaç tane düğümünden oluşur?

Örneğin



1. düğümünden ilerleme başlarsak, döngüye ait ilk düğüm 4. düğüm olur ve döngü üç düğümünden (4, 5 ve 6) oluşur.

Döngüleri bulmanın kolay bir yolu çizgede ilerlemeye başlayıp bütün ziyaret edilmiş düğümleri tutmaktır. Eğer bir düğüm iki defa ziyaret edildiğiye bu düğümün döngüdeki ilk düğüm olduğuna karar verebiliriz. Bu method $O(n)$ zamanda ve $O(n)$ hafızayla çalışır.

Fakat, döngü bulmak için daha iyi algoritmalar var. Her ne kadar bu algoritmaların da zaman karmaşıkları $O(n)$ olsa da toplam kullandıkları hafıza $O(1)$ olur. Eğer n büyük ise bu önemli bir durumdur. Şimdi bu özellikleri barındıran Floyd'un Algoritması'ndan bahsedeceğiz.

Floyd'un Algoritması

Floyd'un Algoritması² a ve b isimli iki işaretçi kullanarak çizgede ilerler. Her iki işaretçi çizgenin başlangıç düğümü olan x düğümünde başlar. Her adımda a 1 adım ileri ve b ise iki adım ileriye gider. Bu işlem işaretçiler birbiriyle buluşana kadar devam eder:

```
a = succ(x);
b = succ(succ(x));
while (a != b) {
    a = succ(a);
    b = succ(succ(b));
}
```

Şu anda a işaretçisi k adım atmışken b işaretçisi $2k$ adım atmıştır ve döngünün uzunluğu k 'yi tam böler. Böylece döngüye ait ilk düğüm a işaretçisini b işaretçisine götürüp birbirleriyle buluştan sonra adım adım hareket edilerek bulunur.

```
a = x;
while (a != b) {
    a = succ(a);
    b = succ(b);
}
first = a;
```

Bundan sonra düğümün uzunluğu aşağıdaki gibi hesaplanır:

```
b = succ(a);
length = 1;
while (a != b) {
    b = succ(b);
    length++;
}
```

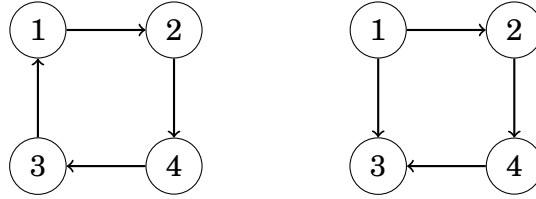
²Algoritmanın fikri [46] kaynağında bahsedilmiştir ve R. W. Floyd'a atfedilmiştir fakat algoritmayı Floyd'un bulup bulmadığı tam bilinmemektedir.

Bölüm 17

Güçlü Bağlanırlık (Strong connectivity)

Yönlü bir çizgede, kenarlar sadece bir yönlü dolaşılabilir yani çizge bağlı olsa bile her düğümden (node) diğer düğümlere rota olacağı garantisi verilemez. Bu nedenden dolayı bağlanırlıktan daha fazla durum gerektiren bir konsept tanımlamak mantıklı olacaktır.

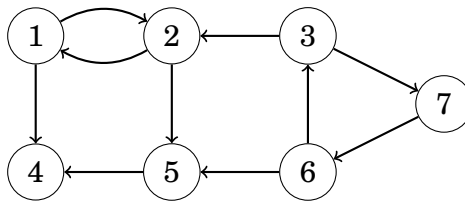
Çizgede eğer her düğümden diğer düğümlere gidilebiliyorsa o zaman bu çizge **güçlü bağlanılmıştır**. Örneğin aşağıdaki resimde soldaki çizge güçlü bağlanılmışken sağdaki öyle değildir.



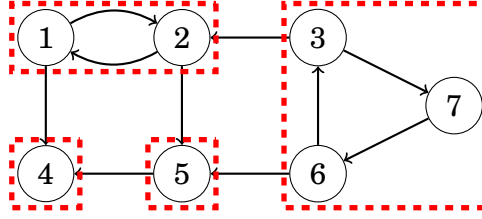
Sağdaki çizge güçlü bağlanılmamıştır çünkü 2. düğümden 1. düğüme herhangi bir rota yoktur.

Çizgenin **güçlü bağlanılmış parçaları (strongly connected components)**, çizgeyi olabildiğince büyük güçlü bağlanılmış bölümlere ayırır. Güçlü bağlanılmış parçalar orijinal çizgenin derin yapısını gösteren bir asiklik (acyclic) (döngü içermeyen) **bileşen çizgesi** oluştururlar.

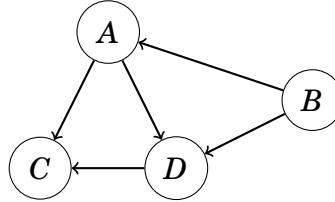
Örneğin aşağıdaki çizgedeki



güçlü bağlanılmış parçalar şekildeki gibidir:



Buna göre oluşan bileşen çizgesi şekildeki gibidir:



Parçalar $A = \{1, 2\}$, $B = \{3, 6, 7\}$, $C = \{4\}$ ve $D = \{5\}$ olur.

Bileşen çizgesi asıklık ve yönlüdür. Böylece orijinal çizgeyi işlemek daha kolay olur çünkü çizge herhangi bir döngü içermez. Böylece topolojik sıralama (topological sort) oluşturup 16. Bölümde anlatılan teknikler gibi dinamik programlama teknikleri kullanılabilir.

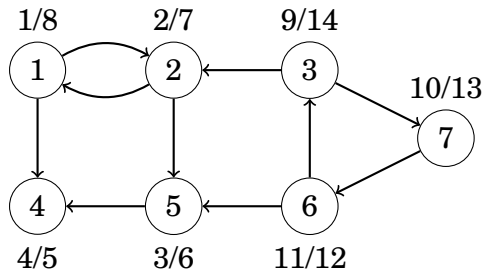
17.1 Kosaraju'nun Algoritması (Kosaraju's Algorithm)

Kosaraju'nun Algoritması¹ yönlü bir çizgede güçlü bağlantılı parçaları bulmaya yaran verimli bir algoritmadır. Algoritma iki tane derinlik öncelikli arama yapar. İlk aramada çizgenin yapısına göre bir düğüm listesi oluşturur ve ikinci aramada da güçlü bağlantılı parçaları oluşturur.

1. Arama

Kosaraju'nun Algoritması'nın ilk evresinde derinlik öncelikli aramanın işleme sırasına göre bir düğüm listesi oluşturur. Algoritma, bu düğümlerden geçer ve her işlenmemiş düğüm için bir derinlik öncelikli arama başlatır. Her düğüm işlendikten sonra listeye eklenir.

Aşağıdaki çizgede, düğümler şekildeki sırada işlenir.



¹[1]e göre S. R. Kosaraju bu algoritmayı 1978'te icat etse de yayınlamamıştır. Aynı algoritma M. Sharir [57] tarafından 1981'de yeniden keşfedilmiştir.

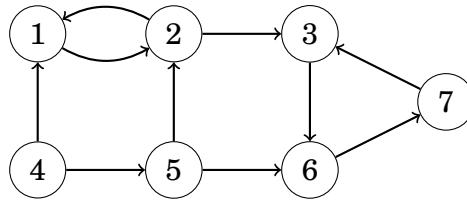
x/y gösterimi düğümü işlemeye x zamanında başlayıp y zamanında işlemeyi bitirdiğini gösterir. Böylece buna göre oluşan liste aşağıdaki gibidir: m

Düğüm	İşleme Zamanı
4	5
5	6
2	7
1	8
6	12
7	13
3	14

2. Arama

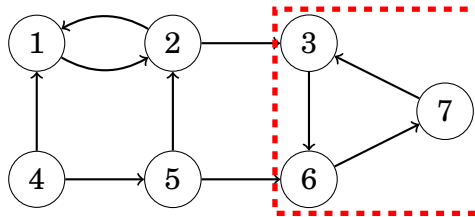
Algoritma, 2. evresinde çizgenin güçlü bağlantılımış parçalarını oluşturur. İlk başta algoritma çizgedeki bütün kenarları tersine çevirir. Bu 2. aramada her seferinde fazladan düğüm içermeyecek güçlü bağlantılımış parça bulmamızı garanti edecek.

Kenarları tersine çevirdikten sonra örnek çizge aşağıdaki gibi olur:



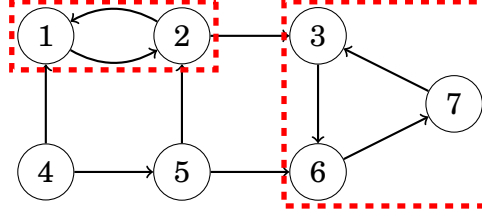
Bundan sonra algoritma 1. aramada oluşturduğu düğüm listesindeki düğümlere *ters* sırada dolaşmaya başlar. Eğer döngü herhangi bir parçaya ait değilse algoritma yeni bir parça oluşturup bu düğümde bir derinlik öncelikli arama başlatır. Bu aramada bulunan bütün düğümleri de bu yeni oluşturulmuş parçaya ekler.

Örneğin, ilk parça 3. düğümde başlıyor:

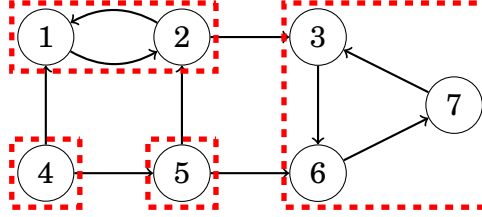


Bütün kenarlar tersine döndürüldüğü için parça çizgenin diğer parçalarına "sızmayacaktır".

Listede bulunan sonraki düğümler 7 ve 6. düğümlerdir ama bu düğümler zaten bir parçaya ait oldukları için yeni parça 1. düğümden başlayacaktır:



En sonda algoritma 5 ve 4. düğümleri işleyerek geriye kalan güçlü bağlantılı parçaları oluşturur:



Algoritmanın zaman karmaşıklığı $O(n + m)$ olur çünkü algoritma iki tane derinlik öncelikli arama yapar.

17.2 2SAT Problemi

Güçlü bağlantırlık aynı zamanda **2SAT problemi** ile de ilişkilidir.². Bu problemde bize

$$(a_1 \vee b_1) \wedge (a_2 \vee b_2) \wedge \cdots \wedge (a_m \vee b_m),$$

a_i ve b_i ya mantıksal değişken (logical variable) (x_1, x_2, \dots, x_n) ya da mantıksal değişkenin karşıtı $(\neg x_1, \neg x_2, \dots, \neg x_n)$ olduğunu gösteren ifadelerdir. " \wedge " ve " \vee " sembolleri mantıksal operatörler olan "ve" ve "veya" ifadelerinin gösterimidir. Buradaki görev, her değişkene öyle bir değer verilmelidir ki formül doğru olmalı veya böyle bir durumun olmayacağını belirtilmesi gerekir.

Örneğin

$$L_1 = (x_2 \vee \neg x_1) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_4)$$

formülü değişkenler aşağıdaki gibi olduğu zaman doğru olur:

$$\begin{cases} x_1 = \text{false} \\ x_2 = \text{false} \\ x_3 = \text{true} \\ x_4 = \text{true} \end{cases}$$

Ama

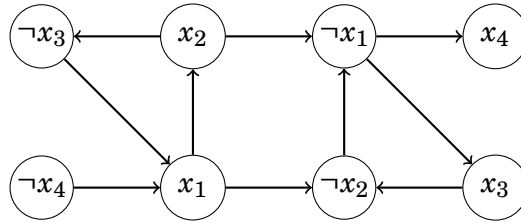
$$L_2 = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_3)$$

²Burada sunulan algoritma [4]te tanıtıldı. Başka bilinen linear çalışan algoritma ise [19] idir ve bu algoritma geri izleme(backtracking) üzerine yapılır.)

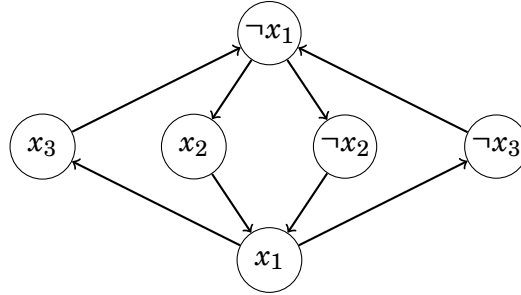
formülü değişkenlere ne değeri verirsek verelim her zaman yanlıştır. Bunun nedeni ise herhangi bir çelişki sağlamayacak x_1 değeri veremeyiz. Eğer x_1 yanlışa, hem x_2 hem de $\neg x_2$ doğru olmalıdır ki bu imkansızdır ve eğer x_1 doğruysa hem x_3 hem de $\neg x_3$ doğru olmalıdır ki bu da imkansızdır.

2SAT problemi düğümleri x_i ve $\neg x_i$ değişkenlerine ve kenarları da değişkenler arasındaki ilişkiye gelecek şekilde bir çizge halinde düşünülebilir. Her $(a_i \vee b_i)$ çifti iki tane kenar oluşturur: $\neg a_i \rightarrow b_i$ ve $\neg b_i \rightarrow a_i$. Bu bize a_i olmazsa b_i değişkenin tutması gerektiğini veya tam tersi olması gerektiği anlamına gelir.

L_1 formülü için oluşan çizge:

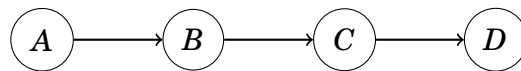


Ve L_2 formülünün çizgesi ise:



şeklinde olur. Çizgelerin yapısı, bize değerleri öyle bir şekilde yerleştirip formülün doğru olup olamayacağını gösterir. Bu durum x_i ve $\neg x_i$ düğümlerinin aynı güçlü bağlantılmış parçada olmadığı sürece doğru olacağını gösterir. Eğer x_i 'den x_i 'ye ve $\neg x_i$ 'den x_i 'ye rota bulunduran öyle düğümler varsa formülün doğru olabilmesi için hem x_i hem de $\neg x_i$ değişkenlerinin doğru olması gerekir ki bu imkansızdır.

L_1 formülünün çizgesinde aynı güçlü bağlantılmış parçada bulunan x_i ve $\neg x_i$ düğümü bulunmadığı için bir çözüm vardır ama L_2 formülünün çizgesinde her düğüm aynı güçlü bağlantılmış parçaya ait olduğu için bir çözüm bulunmamaktadır. Eğer bir çözüm varsa, değişkenlerin değerleri ters topolojik sıralama yapılarak bulunabilir. Her adımda işlenmemiş bileşene kenarı olmayan bileşenleri işleriz. Eğer bileşendeki değişkenlere değer verilmediyse bileşendeki değerlere göre değer verilir ve eğer değerleri varsa da bu değerler aynı kalır. Her değişkene bir değer verilene kadar bu işlem devam eder.



Bileşenler $A = \{\neg x_4\}$, $B = \{x_1, x_2, \neg x_3\}$, $C = \{\neg x_1, \neg x_2, x_3\}$ ve $D = \{x_4\}$ olur. Çözümü yaparken ilk D bileşenini işleriz ve x_4 doğru olur. Sonrasında C bileşenini işleriz

ve x_1 , x_2 yanlış ve x_3 doğru olur. Bütün değişkenlere değer verildiği için kalan A ve B bileşenleri değişken değiştirmez.

Bu method çizgenin özel bir yapısı olduğu için çalışır. Eğer x_i düğümünden x_j düğümüne ve x_j düğümünden $\neg x_j$ düğümüne yol varsa x_i asla doğru olamaz. Bunun nedeni ise de $\neg x_j$ düğümünden $\neg x_i$ düğümüne yol bulunur ve böylece x_i ve x_j ikisi de yanlış olur.

Daha zor bir problem ise **3SAT problemidir**. Bu problemde formülün her parçası $(a_i \vee b_i \vee c_i)$ şeklindedir. Bu soru NP-Hard'dır yani bu soruyu çözebilecek bilinen verimli bir algoritma yoktur.

Bölüm 18

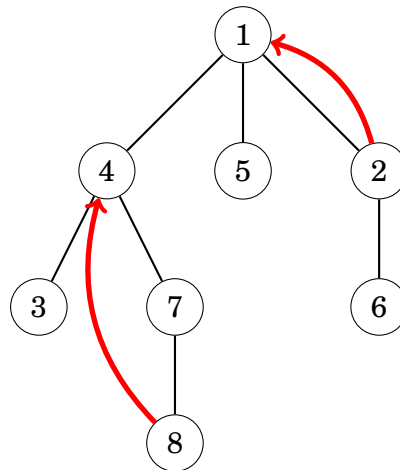
Ağaç Sorguları

Bu bölüm köklü ağaçların alt ağaçları ve yolları üzerinde yapılan sorguları çözme yöntemlerinden bahsedeceğiz. Örneğin bu sorgular:

- düğümün k . atası hangi düğümdür?
- bir ağacın alt ağacında bulunan değerlerin toplamı kaçtır?
- iki düğüm arasında olan yolda bulunan değerlerin toplamı kaçtır?
- iki düğümün en yakın ortak atası hangisidir?

18.1 Ataları Bulmak

Köklü ağaçta bir x düğümünün k . atası, x düğümünden k defa yukarı çıkılarak ulaşacağımız düğümdür. $\text{ancestor}(x, k)$ x düğümünün k . atasını (Eğer bu atası yoksa 0) ifade etsin. Örneğin aşağıdaki ağaçta $\text{ancestor}(2, 1) = 1$ ve $\text{ancestor}(8, 2) = 4$ olur.



Herhangi bir $\text{ancestor}(x, k)$ değerini hesaplamamanın kolay bir yolu vardır. Bu yol ağaçta k adım ilerlemektir. Fakat bu yöntemin zaman karmaşıklığı $O(k)$ olur ve duruma göre yavaş olabilir çünkü ağaç n düğümden oluşuyorsa n tane düğümden oluşan bir zincir olabilir.

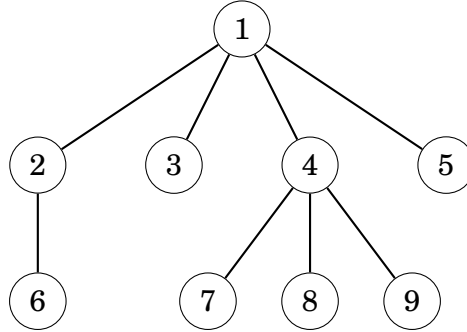
Neyseki Bölüm 16.3'te kullanılan bir yöntemin benzerini kullanarak önceden işleme yöntemi ile $\text{ancestor}(x, k)$ değerini $O(\log k)$ zamanda verimli bir şekilde hesaplayabiliriz:

x	1	2	3	4	5	6	7	8
$\text{ancestor}(x, 1)$	0	1	4	1	1	2	4	7
$\text{ancestor}(x, 2)$	0	0	1	0	0	1	1	4
$\text{ancestor}(x, 4)$	0	0	0	0	0	0	0	0
...								

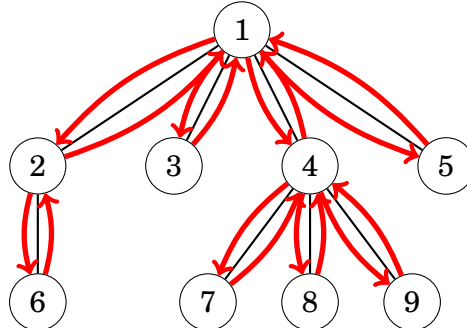
Ön işleme $O(n \log n)$ zaman alır çünkü her düğüm için $O(\log n)$ tane değer hesaplanır. Bundan sonra herhangi bir $\text{ancestor}(x, k)$ değerini k sayısını ikinin kuvvetlerinin toplamı şeklinde göstererek $O(\log k)$ zamanda hesaplayabiliriz.

18.2 Alt Ağaçlar ve Yollar

Ağaç dolaşma dizisi köklü bir ağacın düğümlerini, kök düğümünden başlayan derinlik öncelikli aramada ziyaret etme sırasına göre sıralanmış şekilde sırada tutar. Örneğin



çizgesinde derinlik öncelikli arama aşağıdaki gibidir:



Böylece ağaç dolaşma dizisi aşağıdaki gibi olur:

1	2	6	3	4	7	8	9	5
---	---	---	---	---	---	---	---	---

Alt Ağaç Sorguları

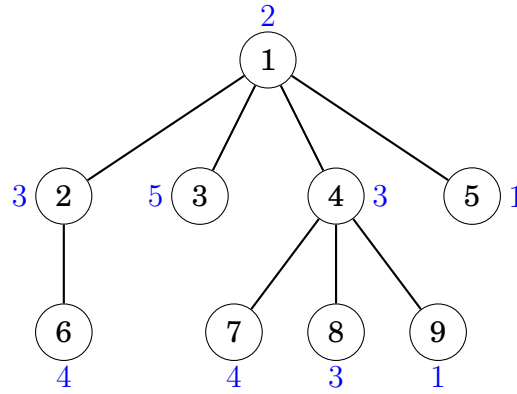
Bir ağacın her alt ağacı ağaç dolaşma dizisinin bir alt dizisine denk gelir. Bu alt dizinin ilk elementi bu alt ağacın kök düğümü olur. Örneğin aşağıdaki alt dizi, 4. düğümünün alt dizisidir:

1	2	6	3	4	7	8	9	5
---	---	---	---	---	---	---	---	---

Bunu kullanarak alt ağaçlarla alakalı sorguları hızlı bir şekilde çözebiliriz. Örnek olarak her düğüme bir değer verildiğini ve amacımızın aşağıdaki sorguları bulabilen şekilde bir kod yazmak olduğunu düşünelim:

- düğümüm değerini güncelle
- düğümün alt ağacında bulunan değerlerin toplamını hesapla

Aşağıdaki ağaçta, mavi sayıların düğümlerin değerleri olduğunu düşünelim. Örneğin 4. düğümün alt ağaç toplamı $3 + 4 + 3 + 1 = 11$ olur.



Buradaki fikir her düğüm için tane 3 değer tutan bir ağaç dolaşma dizisi oluşturmaktır. Bu değerler: düğümün göstergesi, alt ağacın büyüklüğü ve düğümün değeri. Örneğin yukarıdaki ağaç için dizi aşağıdaki gibi olur:

düğüm	1	2	6	3	4	7	8	9	5
alt ağaç büyüklüğü	9	2	1	1	4	1	1	1	1
düğümün değeri	2	3	4	5	3	4	3	1	1

Bu diziyi kullanarak herhangi bir alt ağaçta bulunan değerlerin toplamını ilk başta alt ağacın büyüklüğünü bularak ve sonra düğümlerin değerlerini bularak yapabiliriz. Örneğin 4. düğümün alt ağacında bulunan değerlerin toplamı aşağıdaki gibi bulunur:

düğüm	1	2	6	3	4	7	8	9	5
alt ağaç büyüklüğü	9	2	1	1	4	1	1	1	1
düğümün değeri	2	3	4	5	3	4	3	1	1

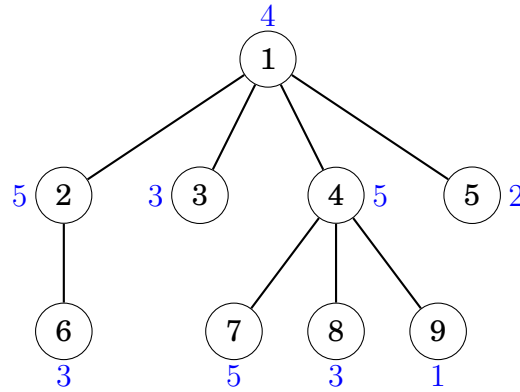
Bu sorguları verimli bir şekilde çözmek için düğümlerin değerlerini ikili indeks (binary indexed) ağacı ile veya bölüm ağacı (segment tree) ile tutmak yeterlidir. Bundan sonra herhangi bir düğümün değerini güncellemeyi ve değerlerin toplamını hesaplamayı $O(\log n)$ sürede yapabiliriz.

Yol Sorguları

Ağaç dolaşma dizileri kullanarak kök düğümünden herhangi bir düğüme olan yolda bulunan değerlerin toplamını kullanarak hesaplayabiliriz. Amacımız aşağıdaki sorguları çözebilen bir çözüm yapmak olsun:

- düğümün değerini değiştirmek
- kök düğümünden bir düğüme giden yolda bulunan düğümlerin toplamını hesaplamak

Örneğin aşağıdaki ağaçta kök düğümünden 7. düğüme olan değerlerin toplamı $4 + 5 + 5 = 14$ olur:



Bu soruyu öncekisi gibi çözebiliriz ama şimdi dizinin son sırasında bulunan her değer, kök düğümünden o düğüme olan yolda bulunan değerlerin toplamına eşittir. Örneğin aşağıdaki dizi yukarıdaki ağaca karşılık gelir:

düğüm	1	2	6	3	4	7	8	9	5
alt ağaç büyüklüğü	9	2	1	1	4	1	1	1	1
yoldaki değerlerin toplamı	4	9	12	7	9	14	12	10	6

Bir düğüm değeri x arttığı zaman, onun alt ağacında bulunan bütün düğümlerin değeri x artar. Örneğin 4. düğümün değeri 1 arttığı zaman dizi aşağıdaki şekilde değişir:

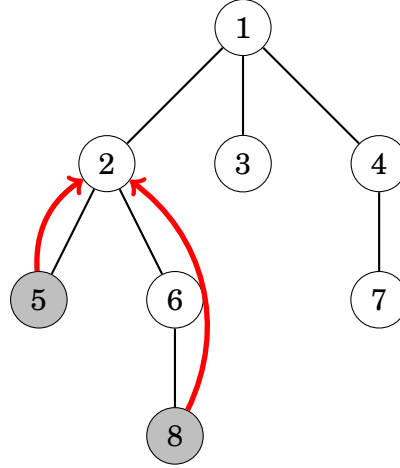
düğüm	1	2	6	3	4	7	8	9	5
alt ağaç büyüklüğü	9	2	1	1	4	1	1	1	1
yoldaki değerlerin toplamı	4	9	12	7	10	15	13	11	6

Böylece iki operasyonu yapabilmek için değerleri bir aralıkta arttırmamız ve tek bir değeri alabiliyor olmamız gerekir. Bu $O(\log n)$ zamanda ikili indeksli ağaç veya bölüm ağacı ile yapılabilir (Daha detaylı bilgi için Bölüm 9.4'e bakın.).

18.3 En Yakın Ortak Ata (Lowest Common Ancestor (LCA))

İki düğümün **en yakın ortak atası** köklü ağaçta, alt ağacında iki düğümü de bulunduran en aşağıdaki düğümdür. Klasik bir problem, iki düğümün en yakın orta atasını bulmamızı isteyen sorguları çözmek olur.

Örneğin aşağıdaki ağaçta, 5 ve 8. düğümün en yakın orta atası 2. düğümdür:



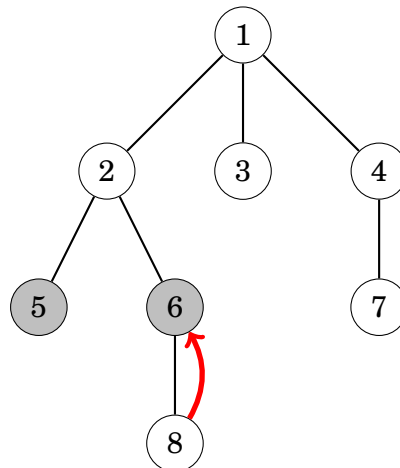
Şimdi iki düğümün en yakın ortak atasını bulmak için iki farklı verimli tekniği kullanabiliriz:

1. Method

Bu soruyu çözmenin bir yolu, ağaçtaki herhangi bir düğümün k . atasını verimli bir şekilde çözebilmemiz ile yapılır. Bunu kullanarak soruyu iki parçaya bölebiliriz.

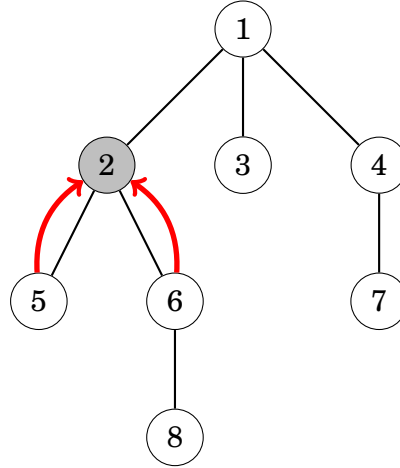
İlk başta en yakın ortak atası bulunmak istenen iki düğümü gösteren İki tane işaretçi tutarız. Sonrasında her seferinde bir tanesini yukarıya taşıyarak ikisinin de aynı seviyede olmasını sağlarız.

Aşağıdaki durumda, ikinci işaretçi bir seviye üste çıkıp 6. düğümü gösterir ki bu 5. düğüm ile aynı seviyededir:



Bundan sonra iki işaretçinin de aynı düğüme ulaşması için gereken minimum adım sayısını hesaplarız. Bundan sonra işaretçilerin göstereceği düğüm, başlangıçtaki iki düğümün en yakın ortak atası olur.

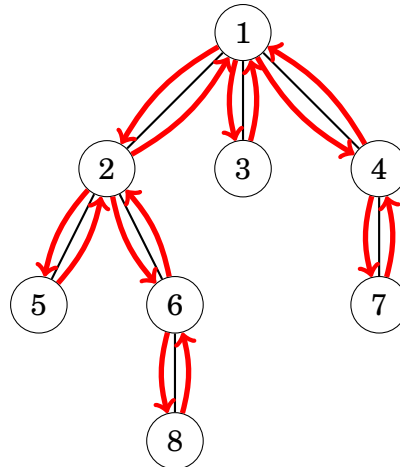
Aşağıdaki durumda iki işaretçiyi de bir adım yukarıya taşıp 2. düğüme getirmek yeterlidir:



Algoritmanın iki parçası da önceden işlenmiş bilgi sayesinde $O(\log n)$ sürede çalışır. Böylece iki düğümün en yakın ortak ata düğümünü $O(\log n)$ sürede bulabiliriz.

2. Method

Soruyu çözmenin başka bir yolu ağaç dolaşma dizisi üzerinden yapılabilir.¹ Yine burada da düğümleri derinlik öncelikli arama ile gezeriz:



Fakat burada önceki ağaç dolaşma dizilerinden daha farklı bir dizi kullanılır: Burada derinlik öncelikli aramada ziyaret edilen her düğüm sadece ilk ziyaretinde

¹Bu en yakın ortak ata algoritması [7] kaynağında tanıtıldı. Bu tekniğe bazen **Euler Turu Tekniği** de denir [66].

değil, *her* ziyaretinde diziye eklenir. Bu yüzden k tane çocuğu olan bir düğüm $k + 1$ defa dizide bulunur. Bu yüzden toplam dizide $2n - 1$ tane düğüm olur.

Dizide iki farklı değer tutarız: düğümün göstergesi ve düğümün ağaçtaki derinliği. Aşağıdaki dizi yukarıdaki ağaca karşılık gelir:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
node id	1	2	5	2	6	8	6	2	1	3	1	4	7	4	1
depth	1	2	3	2	3	4	3	2	1	2	1	2	3	2	1

Şimdi a ve b düğümlerinin en yakın ortak atalarını dizide a ve b düğümleri arasında en az derinliğe sahip (ağaçta en yukarıda olan) düğüm seçilir. Örneğin 5 ve 8. düğümlerinin en yakın ortak atası aşağıdaki gibi bulunur:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
node id	1	2	5	2	6	8	6	2	1	3	1	4	7	4	1
depth	1	2	3	2	3	4	3	2	1	2	1	2	3	2	1

↑

5. düğüm 2. pozisyondayken 8. düğüm de 5. pozisyonadadır. 2...5 aralığı arasında minimum derinliğe sahip olan düğüm 3. pozisyonda olup 2 derinliğinde olan 2. düğümdür.

Böylece iki düğümün en yakın ortak atası, bir minimum sayı bulma sorgusuna eşittir. Dizi statik olduğu için bu tip sorguları $O(1)$ zamanda bulabilmek için $O(n \log n)$ zaman süren ön işleme yapılır.

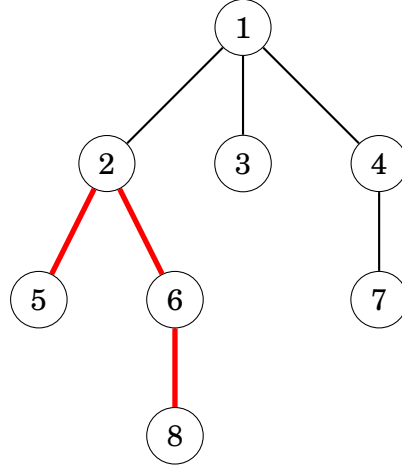
Düğümlerin Mesafeleri

a ve b düğümleri arasında mesafe a düğümünden b düğümüne olan yolun uzaklığına eşittir. Burada problem, düğümler arasında mesafeleri bulmak, en yakın ortak ata bulmaya dönüşüyor.

İlk başta ağacın herhangi bir düğümü kök olarak seçilir. Bundan sonra a ve b düğümleri arasındaki mesafe

$$\text{depth}(a) + \text{depth}(b) - 2 \cdot \text{depth}(c),$$

formülü ile bulunabilir. Burada c , a ve b düğümlerinin en yakın ortak atası olur ve $\text{depth}(s)$ s düğümünün derinliğini gösterir. Örneğin 5 ve 8. düğümleri arasındaki mesafeye bakalım:



5 ve 8. düğümlerinin en yakın ortak atası 2. düğümdür. Bu düğümlerin derinlikleri $\text{depth}(5) = 3$, $\text{depth}(8) = 4$ ve $\text{depth}(2) = 2$ olur ve bu yüzden 5 ve 8. düğüm arasındaki mesafe $3 + 4 - 2 \cdot 2 = 3$ olur.

18.4 Çevrimdışı Algoritmalar

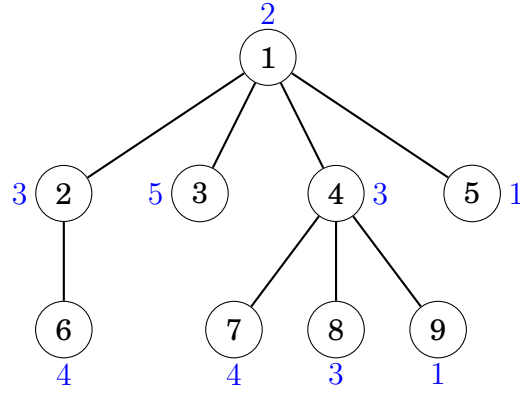
Şu ana kadar ağaç sorguları için *çevrimiçi* algoritmalara baktık. Bu çevrimiçi algoritmalar, sonraki sorgu gelmeden şimdiki sorguyu cevaplandırabilir.

Fakat, çoğu problemde çevrimiçi özellik gerekli değildir. Bu bölümde *çevrimdışı* algoritmalarından bahsedeceğiz. Bu algoritmalarda sorgular herhangi bir sırayla cevaplandırılabilir. Genelde çevrimdışı bir algoritma yapmak çevrimiçi bir algoritma yapmaya göre daha kolaydır.

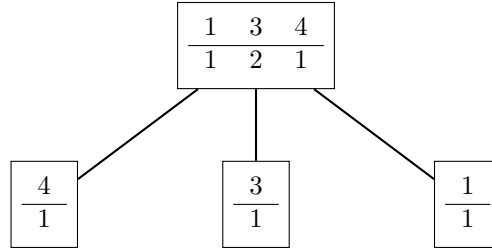
Veri Yapılarını Birleştirmek

Çevrimdışı bir algoritma yapmanın yollarından biri derinlik öncelikli ağaç araması yapıp veri yapılarını düğümlerde tutmaktır. Her s düğümünde $d[s]$ yapısı oluştururuz ve bu yapı s düğümünün çocuklarının veri yapılarına göre şekillenir. Sonra bu yapıyı kullanarak s ile ilgili olan sorgular işlenir.

Örnek olarak şu problemi düşünelim. Bize her düğümünde belirli bir değer olan bir ağaç verildiğini varsayalım. Amacımız " s düğümünün alt ağacında bulunan x değerine sahip düğüm sayısını hesaplamak" sorgularını çözmektir. Örneğin aşağıdaki ağaçta 4. düğümün alt ağacı 3 değerine sahip 2 tane düğüm vardır.



Bu problemde, soruları çözmek için map yapılarını kullanabiliriz. Örneğin 4. düğümün ve çocuklarının mapleri aşağıdaki şekildedir:



Her düğüm için böyle bir veri yapısı oluşturursak verilen bütün sorguları kolay bir şekilde yapabiliriz çünkü bir düğümle alakalı olan bütün sorguları veri yapısını oluşturduktan sonra çözebiliriz. Örneğin yukarıdaki 4. düğümün map yapısı bize alt ağacının 3 değerine sahip iki tane düğüm içerdiğini söylüyor.

Fakat, bütün veri yapılarını en baştan yapmak çok yavaş olur. Bunun yerine her s düğümü için sadece s düğümünün değerini tutan bir başlangıç $d[s]$ veri yapısı oluştururuz. Bundan sonra s düğümünün çocuklarından geçeriz ve $d[s]$ ile s düğümünün çocukları olan u düğümünün $d[u]$ veri yapılarını *birleştiririz*

Örneğin yukarıdaki ağaçta, 4. düğümün mapi aşağıdaki mapleri birleştirerek oluşturulabilir:



Buradaki ilk map 4. düğümün için ilk veri yapısıdır ve diğer üç map 7, 8 ve 9. düğümlere karşılık gelir.

s düğümünü aşağıdaki şekilde birleştirebiliriz: s düğümünün çocuklarından geçeriz ve her u çocuğunda $d[s]$ ile $d[u]$ yapılarını birleştiririz. Fakat bundan önce eğer $d[s]$, $d[u]$ yapısından küçükse $d[s]$ ile $d[u]$ yapılarını birbirleriyle *değiştiririz*. Bunu yaparak her değer ağaç dolaşımı sırasında en fazla $O(\log n)$ defa kopyalanır. Bu durum da algoritmanın verimli olduğunu doğrular.

a ve b veri yapılarının içeriklerini verimli bir şekilde birbiriyle değiştirmek için aşağıdaki kodu kullanabiliriz:

```
swap(a,b);
```

Yukarıdaki fonksiyon a ve b C++ standart kütüphane veri yapısı olduğu sürece sabit bir zamanda çalışmayı garantiler.

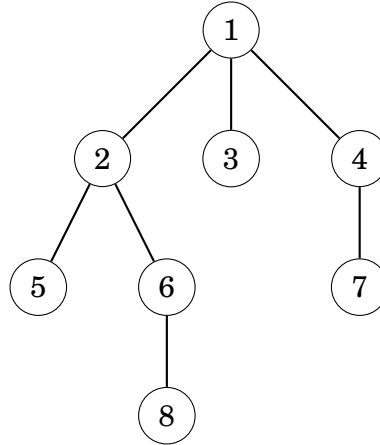
En Yakın Ortak Atalar (Lowest Common Ancestors)

En yakın ortak ataları işlemek için bir tane de çevrimdışı bir algoritma vardır². Bu algoritma union-find (Bölüm 15.2'ye bakın) yapısı üzerine kuruludur. Bu algoritmanın yararı bu bölümün başında bahsedilen algoritmalarından daha kolay koda geçirilebiliyor olmasıdır.

Algoritmada girdi, düğüm çiftleri olarak alınır ve her düğüm çifti için bu düğümlerin en yakın ortak ataları bulunur. Algoritma derinlik öncelikli arama ile ağacı dolaşır ve ayrık düğüm gruplarını bulur. Başta her düğüm ayrı bir gruba aittir. Her grup için o sete ait en yukarıdaki düğümü kaydederiz.

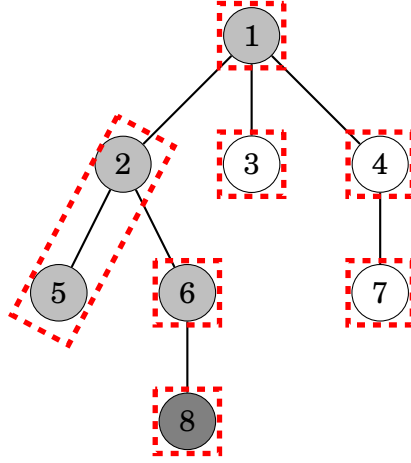
Algoritma bir x düğümünü ziyaret ettiği zaman x ve y düğümlerinin en yakın ortak ataları bulunması gereken bütün y düğümleri bulunur. Eğer y çoktan ziyaret edilmişse algoritma x ve y düğümlerinin en yakın ortak atalarının y düğümünün grubunda bulunan en yukarıdaki düğüm olduğunu söyler. Bundan sonra x düğümünü işledikten sonra algoritma x düğümünü ve ebebeyninin gruplarını birleştirir.

Örneğin aşağıdaki ağaçta $(5,8)$ ve $(2,7)$ düğüm çiftlerinin en yakın ortak atalarını bulmak istiyoruz.

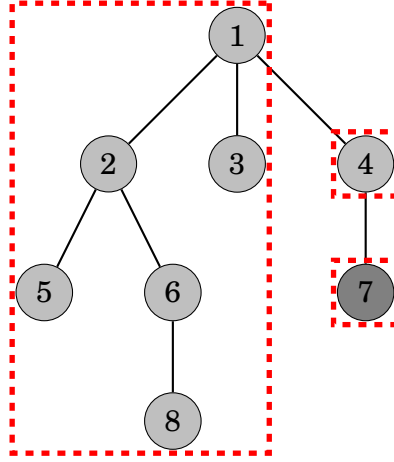


Aşağıdaki ağaçlarda gri düğümler ziyaret edilmiş düğümleri ve çizgili gruplarda bulunan düğümler ise aynı grupta bulunduğunu gösterir. Algoritma 8. düğümü ziyaret ettiği zaman 5. düğümün önceden ziyaret edildiğini ve grubundaki en yüksek düğümün 2. düğüm olduğunu söyler. Bu yüzden 5 ve 8. düğümlerin en yakın ortak atası 2. düğüm olur:

²Bu algoritma R. E. Tarjan tarafından 1979'da yayınlanmıştır [65].



Sonra 7. düğümü ziyaret ettiğimizde 2 ve 7. düğümlerin en yakın atalarının 1. düğüm olduğunu söyler:



Bölüm 19

Yollar ve Devreler

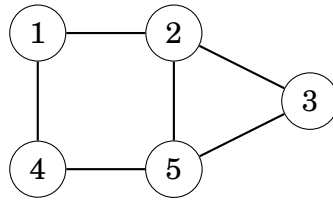
Bu bölüm çizgeler üzerinde iki tip yol üzerinde yoğunlaşıyor:

- An **Euler Yolu (Eulerian Path)** her kenardan tam bir defa geçen yol tipidir.
- A **Hamilton Yolu (Hamiltonian Path)** her düğümden tam bir defa geçen yol tipidir.

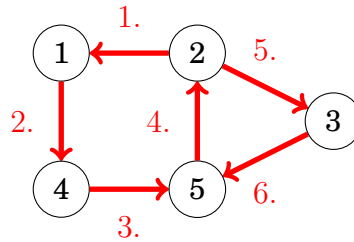
Euler ve Hamilton yolları ilk bakışta benzer konseptte olsalar da onla alakalı problemleri çok farklıdır. Bir çizgenin Euler yolu içerip içermediğini bulmanın çok kolay bir yolu vardır ve eğer varsa da o yolu bulmanın verimli bir algoritması vardır. Fakat Hamilton yolunda işler tam tersidir. Hamilton yolunu NP-hard problemdir ve soruyu çözecek verimli bir algoritma yok.

19.1 Euler Yolu

Euler yolu¹ çizgenin her kenarından tam olarak bir defa geçen yol tipidir. Örneğin

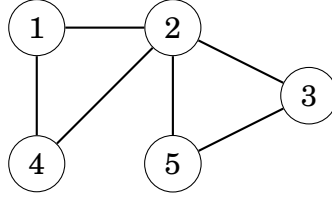


2. düğümden 5. düğüme bir Euler yolu vardır:

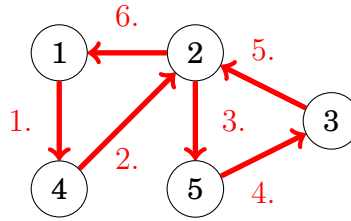


¹L. Euler bu tip yolları 1736 yılında Königsberg Köprüsü sorusunu çözdüğünde araştırmıştır. Bu tam olarak çizge teorisinin ortaya çıkışıdır.

Euler Devresi aynı düğümde başlayıp biten Euler yoludur. Örneğin



1. düğümde başlayıp biten bir Euler devresine sahiptir:



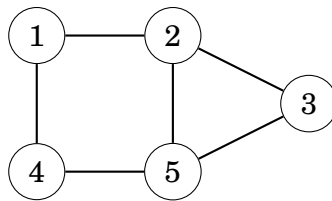
Çizgenin Euler Yolu İçerip İçermemesi

Çizgede Euler yolunun ve devrelerinin olup olmaması düğümlerin derecelerine bağlıdır. Eğer yönsüz bir çizgede bütün kenarlar tek bir parçaya aitse ve

- her düğüm derecesi çift *veya*
- iki düğüm derecesi tek ve diğerlerinin çift

ise bu çizge bir Euler yolu içerir. Eğer her düğümün derecesi çift ise her Euler yolu aynı zamanda bir Euler devresi olur. Fakat iki düğümün derecesi tek ise tek dereceli düğümlerden başlayan ve biten Euler yolları Euler devreleri olamaz.

Örneğin



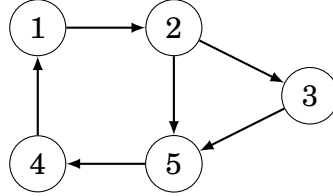
çizgesinde 1, 3 ve 4. düğümlerin derecesi 2'yken 2 ve 5. düğümlerin derecesi 3'tür. Burada tam iki tane düğüme tek derecesi vardır ve 2 ile 5. düğümler arasında Euler yolu vardır fakat çizge Euler devresi içermemektedir.

Yönlü bir çizgede, düğümlerin iç ve dış derecelerine önem veririz. Yönlü bir çizgede her kenarlar aynı parçaya aitse ve

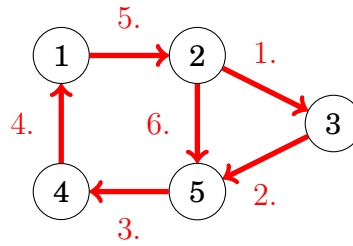
- her düğümde iç derece dış dereceye eşitse *veya*
- bir düğümde iç derece dış dereceden bir büyükse ve başka bir düğümde dış derece iç dereceden bir küçükse ve geri kalan diğer tüm düğümlerde iç derece dış dereceye eşitse

bu çizge Euler yoluna sahiptir. İlk durumda her Euler yolu aynı zamanda bir Euler devresi iken ikinci durumda çizge başlangıç düğümünde dış derecesi iç derecesine göre daha büyük olan ve bitiş düğümünde iç derecesi dış derecesine göre daha büyük olan bir Euler yolu içerir.

Örneğin



1, 3 ve 4. düğümlerin hem iç hem de dış dereceleri 1'ken 2. düğümün iç derecesi 1, dış derecesi 2 ve 5. düğümün iç derecesi 2 ve dış derecesi 1 olur. Bu yüzden çizge 2. düğümden 5. düğüme bir Euler yolu içerir:



Hierholzer'in Algoritması

Hierholzer'in Algoritması² Euler devresi kuramk için verimli bir methoddur. Algoritma birkaç tur çalışır ve her turda devreye yeni kenar ekler. Tabii bu algoritmanın çalıştığı çizgenin Euler devresi içermesi gerekir yoksa Hierholzer'in Algoritması bulamaz.

İlk başta, algoritma çizgenin bazı kenarlarını içeren bir devre oluşturur. Bundan sonra algoritma adım adım alt devre ekleyerek devreyi büyütür. Bu bütün kenarlar devreye eklenene kadar devam eder.

Algoritma devreyi her seferinde devreye ait ve dış komşularında devreye ait olmayan bir düğüm içeren x düğümünü bulur. Algoritma x düğümünden başlayan ve devrede bulunmayan kenarlardan oluşan bir yol oluşturur. Eninde sonunda, bu yol x düğümüne geri dönecektir ve bu da alt devre oluşturacaktır.

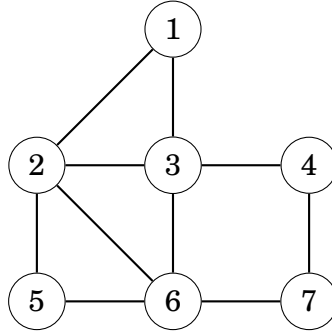
Eğer çizge sadece Euler yolu içeriyorsa Hierholzer'in Algoritmasını devre oluşturdktan sonra çizgeye kenar eklemesi veya çıkarması için kullanabiliriz. Örneğin yönsüz bir çizgede iki tane tek dereceli düğüm arasına ekstra bir yol ekleriz.

Şimdi Hierholzer'in Algoritması'nın yönsüz bir çizgede nasıl Euler devresi oluşturduğuna bakalım.

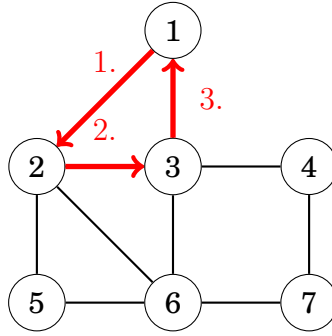
²Bu Algoritma Hierholzer'in ölümünden sonra 1873 yılında yayınlanmıştır [35].

Örnek

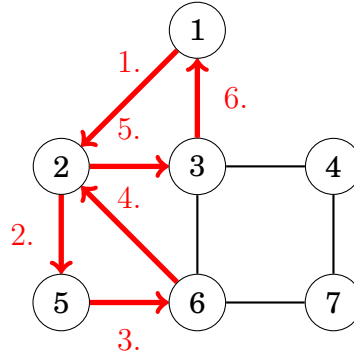
Örneğin aşağıdaki çizgede



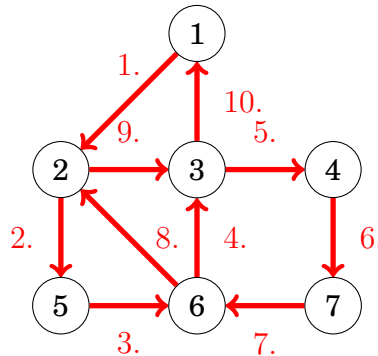
Algoritmanın 1. düğümden bir devre oluşturduğunu varsayalım. Olası bir devre $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$ şeklinde olur:



Bundan sonra algoritma $2 \rightarrow 5 \rightarrow 6 \rightarrow 2$ alt devresini devreye ekler:



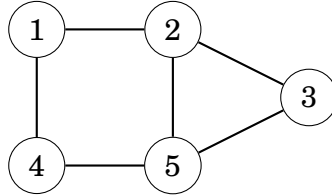
En sonunda algoritma $6 \rightarrow 3 \rightarrow 4 \rightarrow 7 \rightarrow 6$ alt devresini ana devreye ekler:



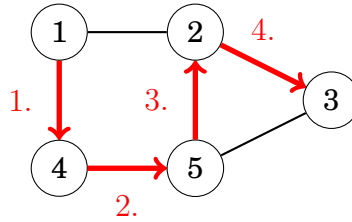
Bütün kenarlar başarıyla devreye eklendiği için başarıyla bir Euler devresi elde etmiş olduk.

19.2 Hamilton Yolları

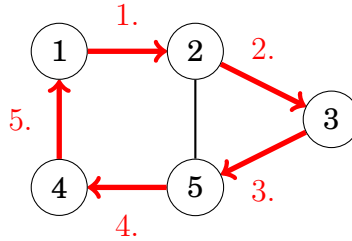
Hamilton yolu çizgedeki her düğümü tam bir kere ziyaret eden yol türüdür. Örneğin



1. düğümden 3. düğüme bir Hamilton yolu içerir:



Eğer Hamilton yolu aynı düğüme başlayıp bitiyorsa bu yol aynı zamanda **Hamilton Devresi** olur. Yukarıdaki çizge aynı zamanda 1. düğümden başlayıp biten bir Hamilton devresi içerir:



Çizgenin Hamilton Yolu İçerip İçermemesi

Çizgenin Hamilton Yolu içerip içermediğini bulmanın verimli bir methodu yoktur ve bu problem NP-hard'dır. Yine de bazı özel durumlarda çizgenin Hamilton yolu içerdiğini kesinlikle bilebiliriz.

Eğer çizge tam çizgeyse yani her iki düğüm arasında kenar varsa bu çizge aynı zamanda Hamilton yolu da içerir. Daha iyi sonuçlar elde edilmiş oluyor:

- **Dirac'ın Teoremi:** Eğer her düğümün derecesi en az $n/2$ ise çizge Hamilton yolu içerir.
- **Ore'nin Teoremi:** Eğer her komşu olmayan düğüm çiftlerinin derece toplamı en az n oluyorsa çizge Hamilton yolu içerir.

Bu teoremlerin ortak bir özelliği ise bu teoremlerin çizge *fazla* sayıda kenar içeriyorsa bu teoremler Hamilton yolu bulunduğunu garantiler. Bu mantıklıdır çünkü çizge daha fazla kenar içeriyorsa Hamilton yolu oluşturmanın daha fazla ihtimali olur.

Hamilton Yolunun Oluşturulması

Çizgede Hamilton yolunun bulunup bulunmadığını kontrol etmenin verimli bir yolu olmadığı için yolu oluşturmanın da verimli bir yöntemi yoktur çünkü olsaydı yolu oluşturmaya çalışıp yolun bulunup bulunmadığına bakardık.

Hamilton yolunu aramanın basit bir yolu oluşturulabilecek bütün yollara geri izleme yöntemi ile bakmaktır. Bu tip bir algoritmanın zaman karmaşıklığı $O(n!)$ olur çünkü n tane düğümün $n!$ tane oluşturulabilecek sırası vardır.

Daha verimli bir çözüm yolu dinamik programlama kullanmaktır (Bölüm 10.5'e bakın). Buradaki fikir $\text{possible}(S, x)$ fonksiyonunun değerlerini hesaplamaktır. Burada S bir düğüm alt kümesi olurken x ise bu düğümlerden biridir. Bu fonksiyon S düğümlerinde dolaşp x düğümünde biten bir Hamilton yolu bulunup bulunmadığını gösterir. Bu tip bir algoritma $O(2^n n^2)$ zamanda çalışabilir.

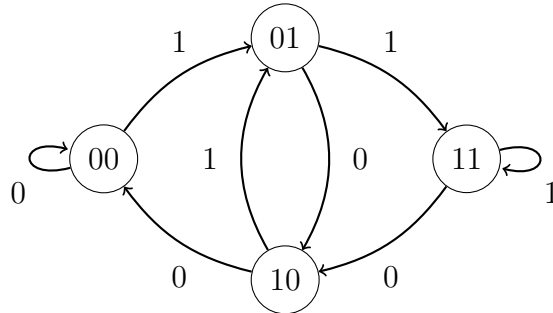
19.3 De Bruijn Dizileri

De Bruijn dizisi her n uzunluğundaki alt yazıyı (substring) tam bir kere içeren k karakterden oluşan bir yazıdır. Bu tip yazılar $k^n + n - 1$ karakterden oluşur. Örneğin $n = 3$ ve $k = 2$ olduğu zaman örnek bir De Bruijn dizisi

0001011100

olur. Bu dizide bulunan alt yazılar üç tane bitin bütün kombinasyonlarını içerir: 000, 001, 010, 011, 100, 101, 110 and 111.

Böylece her De Bruijn sırasının çizgede bir Euler yoluna karşılık geldiği görülüyor. Buradaki fikir, her düğümü $n - 1$ karakterden oluşan bir yazı barındıran bir çizge oluşturmaktır. Her kenar yazıya bir karakter ekler. Aşağıdaki çizgede yukarıdaki diziyi sağlar:



Bu çizgedeki Euler yolu, bütün n uzunluğundaki yazıları kapsayan bir yazıya denk gelir. Yazı başlangıç düğümünün karakterlerini ve kenarların bütün karakterlerini içerir. Başlangıç düğümü $n - 1$ karakter ve kenarlarda toplam k^n karakter vardır. Bu yüzden bu tip bir yazının uzunluğu $k^n + n - 1$ olur.

19.4 Atın Turları (Knight's Tours)

Atın turu bir satranç atının $n \times n$ satranç tahtasında atın her kareyi tam bir ziyaret etmek şartıyla yapabileceği hareketler dizisidir. Eğer at en sonda başladığı kareye dönüyorsa *kapalı* tur ve eğer dönmüyorsa *açık* tur denir.

Örneğin 5×5 satranç tahtasındaki bir açık at turuna bakalım:

1	4	11	16	25
12	17	2	5	10
3	20	7	24	15
18	13	22	9	6
21	8	19	14	23

Bir at turu aslında kareleri düğüm şeklinde görürsek bir Hamilton yoluna karşılık gelir. Burada iki düğüm eğer at, satranç kurallarına göre gidip gelebiliyorsa birbirine kenar ile bağlıdır.

At turu oluşturmanın doğal bir yolu geri izleme kullanmaktır. Aramanın verimli olması için *sezgisel* (heuristics) bir şekilde atı tam bir tur bulmak konusunda hızlı bir şekilde yönetmek gerekir.

Warnsdorf'un Kuralı

Warnsdorf'un Kuralı bir at turu bulmak için kolay bir sezgisel yaklaşımdır³. Bu kuralı kullanarak büyük bir tahta üzerinde verimli bir şekilde tur oluşturmak mümkündür. Buradaki fikir, atı her seferinde olası hareket sayısı en *aza* sahip kareye götürmektir.

Örneğin aşağıdaki durumda, atın hareket edebileceği beş farklı kare vardır (*a...e* kareleri):

1				<i>a</i>
		2		
<i>b</i>				<i>e</i>
	<i>c</i>		<i>d</i>	

Bu durumda Warnsdorf'un kuralı atı *a* karesine hareket ettirir çünkü bu hareketten sonra olası tek bir hareketi vardır. Diğer karelerde at 3 farklı yere gidebilir.

³Bu sezgisel yaklaşım Warnsdorf'un kitabında [69] 1823'te yayınlanmıştır. At turu bulmak için başka polinom algoritmalar da [52] vardır fakat onlar daha karışıktır.

Bölüm 20

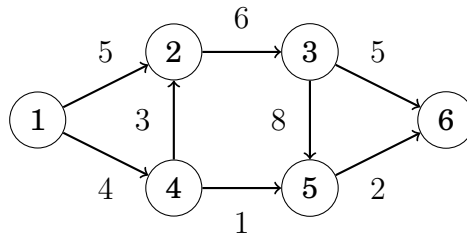
Akışlar (Flows) ve Kesimler (Cuts)

Bu bölümde aşağıdaki iki soruya odaklanacağız:

- **Bir maksimum akış bulmak (maximum flow):** Bir düğümden diğer bir düğüme geçirebileceğimiz en fazla akış ne kadardır? What is the maximum amount of flow we can send from a node to another node?
- **Bir minimum kesim bulmak (minimum cut):** Çizgenin iki düğümünü ayıran en küçük ağırlığa sahip kenarlar grubu nedir?

İki soru için girdi yönlü ve ağırlıklı özel iki düğüm içeren bir çizgedir: *kaynak* düğüm kendisine gelen kenar içermeyen düğümken *musluk* düğümü ise kendisinden çıkan kenar içermeyen düğümdür.

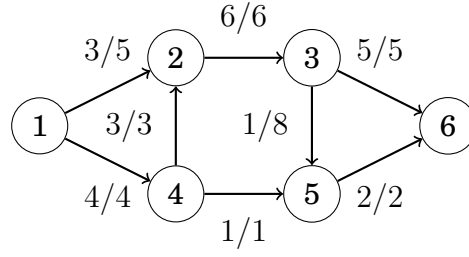
Örneğin, aşağıdaki çizgede 1. düğüm kaynak düğümken 6. düğüm musluk düğümdür:



Maksimum Akış

Maksimum akış probleminde amacımız kaynak düğümden musluk düğüme olabildiği kadar akış göndermektir. Her kenarın ağırlığı o kenardan geçebilecek maksimum akış miktarını gösterir. Her ara düğümde, giren ve çıkan akış eşit olmalıdır.

Örneğin yukarıdaki çizgeden maksimum geçen akış 7'dir. Aşağıdaki resim çözüm rotasını gösterir:

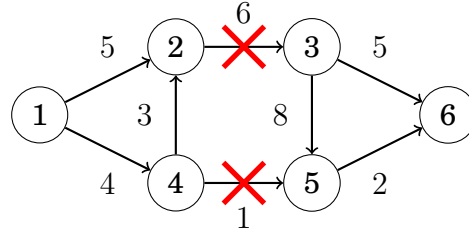


v/k gösterimi kapasitesi k olan kenardan v miktarda akış geçtiğini gösterir. Akışın büyüklüğü 7'dir çünkü kaynak $3 + 4$ miktarda akış gönderir ve musluk $5 + 2$ miktarda akış alır. Bu akışın maksimum olduğunu görmek kolaydır çünkü musluğa gelen kenarların toplam kapasitesi 7 olur.

Minimum Kesim

Minimum kesim probleminde amacımız, çizgeden birkaç kenar çıkardığımızda kaynaktan musluğa herhangi bir yol bulunmamaktadır. Bu çıkarılan kenarların toplam ağırlığı minimumdur.

Yukarıdaki çizgenin minimum kesimi 7 olur. $2 \rightarrow 3$ ve $4 \rightarrow 5$ kenarlarını çıkarmak yeterlidir:



Kenarları çıkardıktan sonra kaynaktan musluğa herhangi bir yol bulunmamaktadır. Kesimin ağırlığı 7'dir çünkü çıkarılan kenarların ağırlığı 6 ve 1'dir. Bu kesim minimumdur çünkü çizgeden istediğimiz şekilde kenar çıkardığımızda toplam ağırlık 7 olmaz.

Bir çizgenin maksimum akışı ile minimum kesiminin aynı olması bir rastlantı değildir. Bu iki *her zaman* eşit büyüklüktedir yani bu konseptler paranın iki yüzü gibidir:

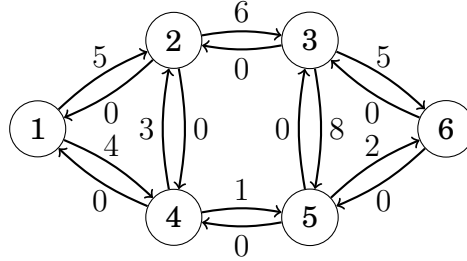
Şimdi çizgenin maksimum akışı ile minimum kesimini bulmaya sağlayacak olan Ford-Fulkerson algoritmasına bakalım. Bu algoritma bize *neden* maksimum akışı ile minimum kesimin aynı olduğunu anlamamıza yardımcı olacak.

20.1 Ford–Fulkerson Algoritması

Ford–Fulkerson Algoritması [25] çizgedeki maksimum akışı bulur. Algoritma boş bir akış ile başlar ve her adımda kaynaktan musluğa daha fazla akış sağlayan bir yol vardır. Sonunda algoritma akışı daha fazla arttıramadığı zaman maksimum akış bulunmuş olur.

Algoritma çizgelerin özel bir gösterimini kullanır. Burada her kenarın zıttı olan bir kenarı da vardır. Her kenarın ağırlığı o kenardan ne kadar akış geçirebileceğimizi gösterir. Algoritmanın başında her kenarın ağırlığı orijinal kenarın ağırlığı olur ve ters kenarın her ağırlığı 0'dır.

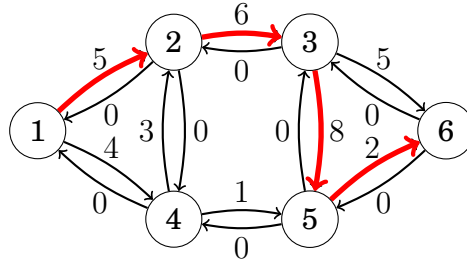
Örnek çizgenin yeni gösterimi aşağıdaki gibidir:



Algoritmanın Açıklaması

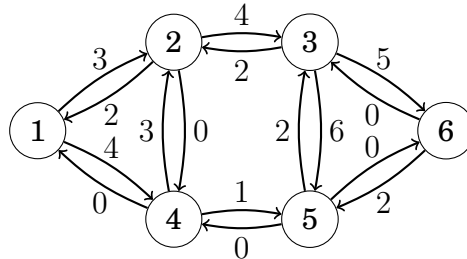
Ford-Fulkerson algoritması birkaç turdan oluşur. Her turda algoritma kaynaktan musluğa hepsi pozitif ağırlığa sahip kenarlardan oluşmuş bir yol bulur. Eğer birden çok yol varsa bu yollardan herhangi birini seçebilir.

Örneğin aşağıdaki yolu seçtiğimizi varsayalım:



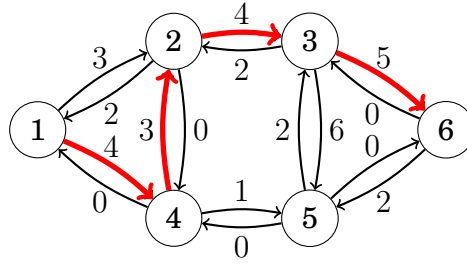
x 'in en küçük kenar ağırlığı olduğu yolu seçtikten sonra akışı x birim arttırabiliriz. Buna beraber yoldaki her kenarın ağırlığı x azalır ve her ters kenarın ağırlığı x arttırır.

Yukarıdaki yoldaki kenarların ağırlıkları 5, 6, 8 ve 2 olur. En küçük ağırlık 2'dir ve akış 2 birim artar. Yeni çizge aşağıdaki gibidir:



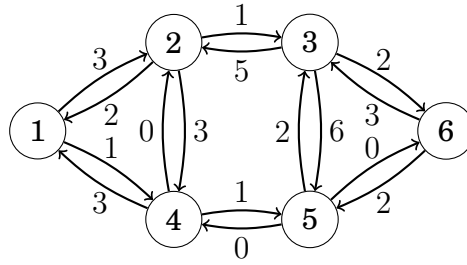
Burada akışı arttırmak, gelecekte kenarlardan geçebilecek akış miktarını azaltır. Diğer tarafta akış, çizgenin ters kenarlarını kullanarak iptal edilebilir. Bunun olması için başka bir rotayı kullanmanın daha yararlı olması gerekir.

Algoritma kaynaktan musluğa pozitif kenarlardan oluşan bir yol olduğu sürece akışı arttırmaktaydı. Şimdi örnekte, sonraki yolumuz aşağıdaki gibi olabilir:

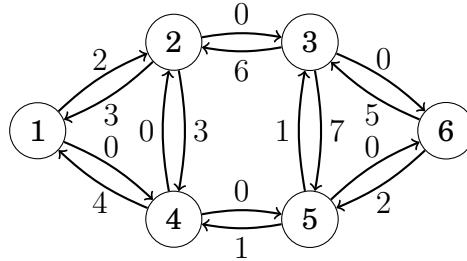


Yoldaki minimum kenar ağırlığı 3'tür yani şu anki yol, akışı 3 birim arttırır ve yol işlendikten sonra gelen toplam akış 5 olur.

Yeni çizge aşağıdaki gibi olur:



Maksimum akışa sahip olmadan önce 2 tura daha ihtiyacımız vardır. Örneğin $1 \rightarrow 2 \rightarrow 3 \rightarrow 6$ ve $1 \rightarrow 4 \rightarrow 5 \rightarrow 3 \rightarrow 6$ yollarını seçebiliriz. En son çizge şekildeki gibi olur:



Akışı daha fazla arttırmak mümkün değildir çünkü kaynaktan musluğa pozitif kenar ağırlıklardan yol bulunmamaktadır. Bu yüzden algoritma biter ve maksimum akış 7 olur. It is not possible to increase the flow anymore, because there is no path from the source to the sink with positive edge weights. Hence, the algorithm terminates and the maximum flow is 7.

Yol Bulmak

Ford-Fulkerson algoritma akışı arttırmak için yolları nasıl seçeceğimizi söylemez. Bu durumda algoritma eninde sonunda maksimum akışı bulur ve algoritma biter. Fakat, algoritmanın verimliliği yolların nasıl seçileceğine göre değişir.

Yol bulmanın kolay bir yolu derinlik öncelikli arama kullanmaktır. Bu genelde iyi çalışır ama en kötü durumda her yol akışı 1 birim arttırır ve algoritma yavaş çalışır. Neyseki bu durumu önlemek için aşağıdaki algoritmalarından birini kullanabiliriz:

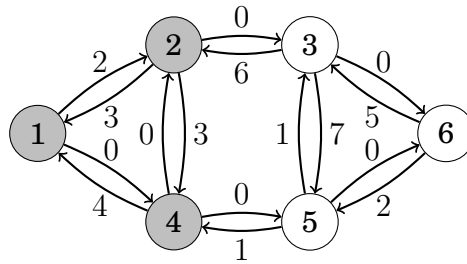
Edmonds–Karp algoritması [18] her seferinde kenar sayısı en az olan yolu seçer. Yolu bulmak için, derinlik öncelikli aramayı kullanmak yerine genişlik öncelikli arama ile yapmak mümkündür. Bu akışı çok hızlı arttıracığını kanıtlar ve bu algoritmanın zaman karmaşıklığı $O(m^2n)$ olur.

Ölçekleme algoritması [2] derinlik öncelikli arama kullanır ve ağırlığı belirli bir eşiği geçen kenarlardan oluşan yolları arar. Başta eşik değeri yüksek bir sayıdır, örneğin çizgedeki bütün kenarların toplamı olur. Bir yol bulunamadığı zaman eşik değeri her seferinde 2'ye bölünür. Bu algoritmanın zaman karmaşıklığı $O(m^2 \log c)$ olur ve c ilk eşik değeridir.

Pratikte, ölçekleme algoritmasını koda dökmek daha iyidir çünkü derinlik öncelikli arama yolları bulmak için kullanılabilir. Her iki algoritma da programlama yarışmalarında çıkan problemler için yeterince verimlidir.

Minimum Kesimler

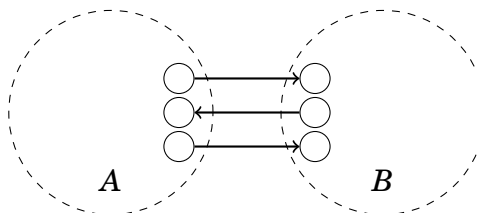
Ford-Fulkerson algoritma maksimum akışı bulduğu zaman aynı zamanda minimum kesimi de bulmuş olur. A grubu, kaynak düğümden pozitif ağırlıklı kenarlarla varılabilecek düğümleri tutar. Aşağıdaki çizgedeki A , 1, 2 ve 4. düğümü içerir:



Şimdi, minimum kesim orijinal çizgede A grubundaki bir düğümden başlayıp A düğümleri harici bir düğümden biten kenarlardan oluşur. Bu kapasite maksimum akışta fullenir. Yukarıdaki çizgede $2 \rightarrow 3$ ve $4 \rightarrow 5$ kenarları $6 + 1 = 7$ minimum kesime karşılık gelir.

Peki neden algoritmanın oluşturduğu akış maksimumken kesim neden minimum olur? Bunun nedeni ise bir akış, çizgedeki herhangi bir kesimin ağırlığından daha büyük olamaz. Bu yüzden bir akış ve kesim eşit olduğu zaman bunlar maksimum akış ile minimum kesim olurlar.

Çizgede kaynağı A 'ya ait olup musluğu B 'ye ait olan herhangi bir kesimi düşünelim ve bu gruplar arasında birkaç tane kenar vardır:



Kesimin ağırlığı A 'dan B 'ye giden kenarların toplamıdır. Bu akış için bir üst sınırdır çünkü akış A 'dan B 'ye işlenir. Böylece, maksimum akışın büyüklüğü çizgedeki herhangi bir çizgeden daha küçük veya eşittir.

Diğer tarafta, Ford-Fulkerson algoritması çizgedeki herhangi bir kesimin olabileceği kadar büyük olan bir akış oluşturur. Böylece akış maksimum akış olmalıdır ve kesim minimum kesim olmalıdır.

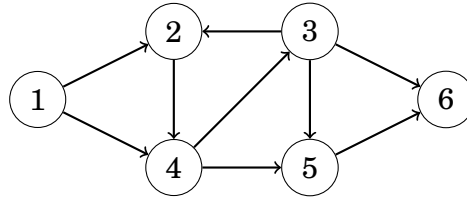
20.2 Ayırık Yollar

Çoğu çizge problemi problemi maksimum akış problemine dönüştürülerek çözülebilir. İlk örnek problem aşağıdaki gibidir: Kaynak ve musluğa sahip olan bir yönlü çizge verildiğini düşünelim ve amacımız kaynaktan musluğa maksimum ayırık yol sayısını bulmaktır.

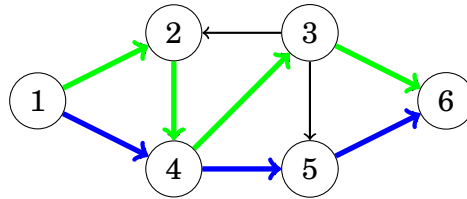
Kenar Ayırık Yollar

İlk başta kaynaktan musluğa olan maksimum **kenar ayırık yol** sayısını bulmaktır. Yani her kenarın en fazla bir yolda bulunduğu yollar grubu oluşturmamız gerekir.

Örneğin aşağıdaki çizgede:



Çizgede maksimum kenar ayırık yol sayısı 2'dir. $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 6$ ve $1 \rightarrow 4 \rightarrow 5 \rightarrow 6$ yollarını seçebiliriz:

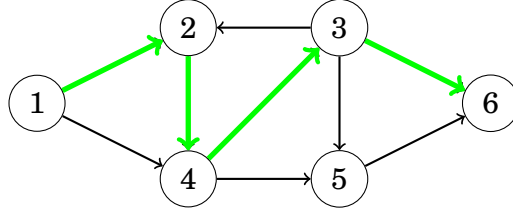


Her kenarın kapasitesinin 1 olduğu durumda maksimum kenar ayırık yol sayısı ile maksimum akışın aynı olduğunu görüyoruz. Maksimum akış oluşturulduktan sonra kenar ayırık yol sayısı açgözlü yöntem ile kaynak düğümden musluk düğüme olan yolları takip ederek bulunabilir.

Düğüm Ayırık Yollar

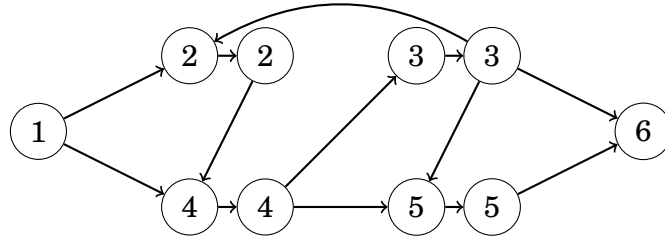
Şimdi başka bir probleme bakalım. Problem, kaynak düğümden musluk düğüme olan en fazla sayıda **düğüm ayırık yol** bulmaktır. Bu problemde, kaynak ve musluk düğüm hariç diğer bütün düğümler en fazla bir yolda bulunabilir. Düğüm ayırık yol sayısı kenar ayırık yol sayısından daha az olabilir.

Örneğin önceki çizgede maksimum düğüm ayırık yol sayısı 1'dir:

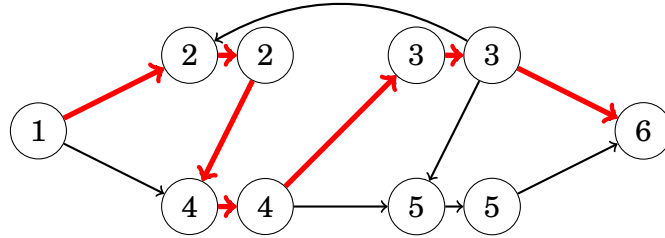


Bu soruyu aynı zamanda maksimum akış sorusuna çevirebiliriz. Her düğüm en fazla bir yolda bulunabildiği için düğümlerden geçen akış miktarı sınırlandırılmalıdır. Bunu yapmanın standart bir yolu her düğümü iki düğüme bölüp ilk düğüm orijinal düğüme gelen kenarlara sahipken ikinci düğüm de orijinal düğümden çıkan kenarlara sahiptir. Bununla beraber 1. düğümden 2. düğüme de bir kenar vardır.

Örneğin, çizge aşağıdaki gibi olur:



Çizgenin maksimum akışı aşağıdaki gibi olur:



Böylece kaynak düğümden musluk düğüme maksimum düğüm ayrık yol sayısı 1'dir.

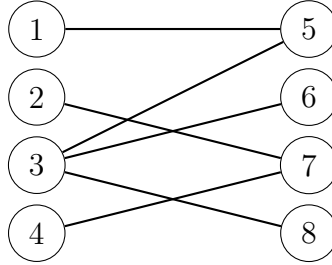
20.3 Maksimum Eşleşme

Makimum eşleşme problemi, yönsüz bir çizgede en büyük olan düğüm çifti kümesi bulmaktır. Bu kümede her düğüm çifti bir kenarla bağlıdır ve her düğüm en fazla bir çifte aittir.

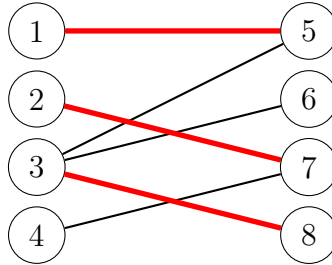
Genel çizgelerde maksimum eşleşmeleri bulmak için polinom zamanlı algoritmalar kullanılabilir. Ama bu tip algoritmalar karışık ve nadiren programlama yarışmalarında kullanılır. Fakat, iki parçalı çizgelerde maksimum eşleşmeyi bulmak daha kolaydır çünkü soruyu maksimum akış probleme dönüştürülebilir.

Maksimum Eşleşmeleri Bulmak

İki parçalı çizgedeki bütün kenarların sol gruptan sağ gruba gidecek şekilde çizgedeki düğümleri iki gruba bölebiliriz. Örneğin, iki parçalı çizgede $\{1,2,3,4\}$ ve $\{5,6,7,8\}$ grupları vardır.

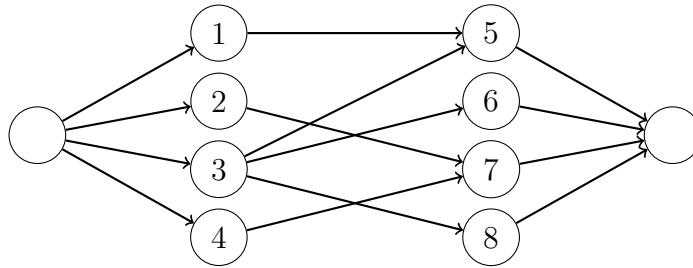


Bu çizgenin maksimum eşleşmesi 3'tür.

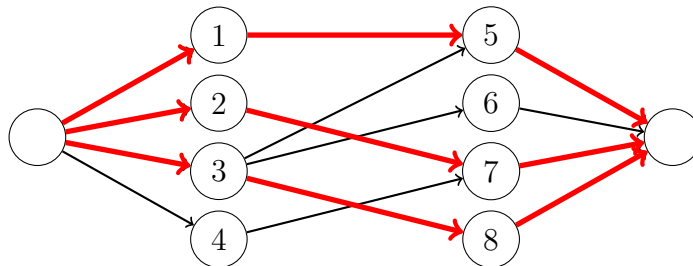


İki parçalı maksimum eşleşme problemi maksimum akış problemine çevirebiliriz. Bunun için çizgeye kaynak ve musluk düğümleri eklemek gerekir. Kaynak düğümünden her sol düğüme ve her sağ düğümden musluğa kenar ekleyebiliriz. Bunda sonra çizgenin maksimum akışının büyüklüğü, orijinal çizgenin maksimum eşleşme büyüklüğüne eşittir.

Örneğin, yukarıdaki çizge aşağıdaki şekilde değiştirilebilir:



Çizgenin maksimum akışı aşağıdaki gibidir:

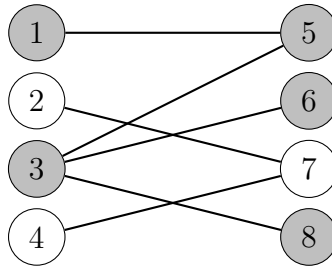


Hall'ın Teoremi

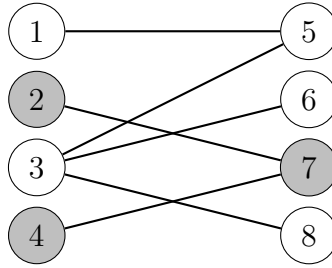
Hall'ın Teoremi bir iki parçalı çizgede bütün sol veya sağ düğümleri kapsayan bir eşleşme bulunup bulunmadığını bulur. Eğer sol ve sağ düğüm sayısı aynıysa, Hall'ın Teoremi bize çizgedeki bütün düğümleri içeren bir **mükemmel eşleşme** yapıp yapılamayacağını söyler.

Diyelim ki bütün sol düğümleri içeren bir eşleşme bulmak istiyoruz. X sol düğümlerden oluşan herhangi bir grup olsun ve $f(X)$ komşularının grupları olsun. Hall'ın Teoremi'ne göre bütün sol düğümleri içeren bir eşleşme olması için her X için $|X| \leq |f(X)|$ şartı sağlanmalıdır.

Hall'ın Teoremi'ni örnek çizgede bakalım. İlk $f(X) = \{5, 6, 8\}$ 'i veren bir $X = \{1, 3\}$ olsun.



Hall'ın Teoremi'nin şartı sağlanıyor çünkü $|X| = 2$ ve $|f(X)| = 3$ idir. Sonra $f(X) = \{7\}$ 'i veren bir $X = \{2, 4\}$ olsun.



Bu durumda $|X| = 2$ ve $|f(X)| = 1$ olur ve bu yüzden Hall'ın Teoremi'nin şartı sağlanmamış oluyor. Bu da bu çizgiden mükemmel eşleşme oluşturmanın mümkün olmadığını gösteriyor. Bu sonuç ilginç değil çünkü zaten çizgenin maksimum eşleşmesi 4 olmayıp 3'tür.

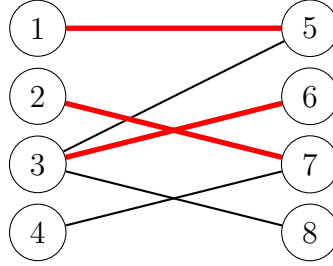
Eğer Hall'ın Teoremi'nin şartı sağlanmazsa X grubu *neden* böyle bir eşleşme oluşmayacağını gösterir. X grubu $f(X)$ 'den daha fazla düğüm içerdiği için X içinde her düğüm için bir çift yoktur. Örneğin yukarıdaki çizgede, hem 2 hem 4. düğümler 7. düğümle bağlanmalıdır ki bu imkansızdır.

König'in Teoremi

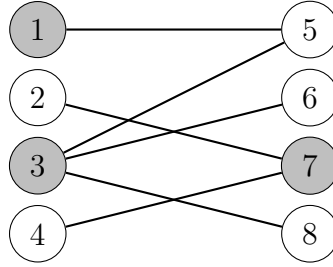
Bir çizgenin **minimum düğüm kaplaması** (minimum node cover), çizgedeki her kenarın bir bitiş noktasının bu gruba ait olan minimum büyüklükteki düğüm grubudur. Genel çizgede minimum düğüm kaplamasını bulmak NP-hard problemdir. Fakat çizge iki parçalı olursa **König'in Teoremi** bize minimum düğüm

kaplaması ile maksimum eşleşmenin büyüklüğü hep aynıdır. Böylece minimum düğümü kaplama büyüklüğünü hesaplamak için maksimum akış algoritmasını kullanabiliriz.

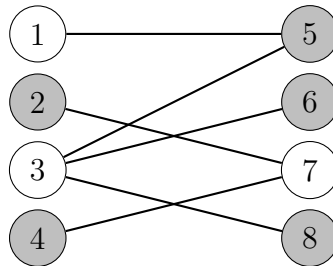
Maksimum eşleşmesi 3 olan aşağıdaki çizgeye bakalım:



König'in Teoremi bize minimum düğüm kaplamasının da 3 büyüklüğünde olduğunu söyler. Bu tip bir kaplama aşağıdaki gibi olur:



Minimum düğüm kaplamasına ait *olmayan* düğümler bir **maksimum bağımsız gruba** (maximum independent set) aittir. Bu, içeriğindeki herhangi iki düğümün birbirine kenarla bağlanmadığı en büyük düğüm grubudur. Yine, genel çizgede maksimum bağımsız grup bulmak NP-hard problemdir ama iki parçalı çizgede König'in Teoremi'ni kullanarak verimli bir şekilde çözülebilir. Örnek çizgede, maksimum bağımsız grup aşağıdaki gibidir:

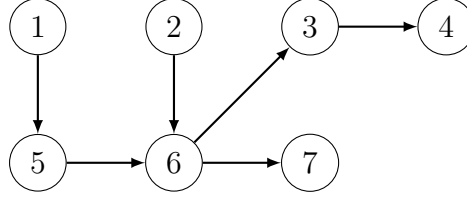


20.4 Yol Kaplaması

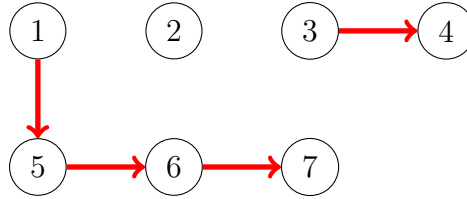
Yol kaplaması çizgede bulunan yollardan oluşan bir gruptur. Çizgedeki her düğüm bu yollardan en az birine aittir. Yönlü asiklik çizgelerde minimum yol kaplama problemini başka bir çizgede maksimum akış problemine çevirebiliriz.

Düğüm Ayırık Yol Kaplaması

Düğüm ayırık yol kaplamasında her düğüm bir yola aittir. Örneğin aşağıdaki çizgede:



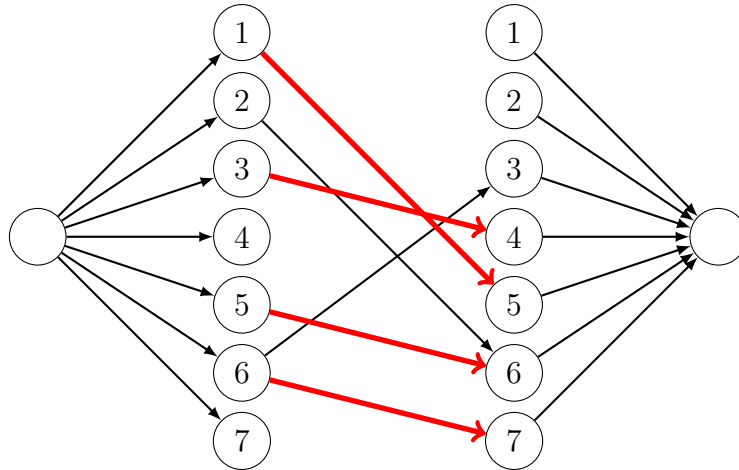
Bu çizgede minimum düğüm ayırık yol kaplaması üç yoldan oluşur. Örneğin aşağıdaki yolları seçebiliriz:



Bu yollardan biri sadece 2. düğümü içerir yani hiçbir kenar içermeyen bir yol bulunabilir.

Minimum düğüm ayırık yol kaplamasını *eşleşen çizge* oluşturarak bulabiliriz. Burada orijinal çizgenin her düğümü iki düğümle gösterilir: Sol düğüm ve sağ düğüm. Sol düğümünden sağ düğüme kenar olabilmesi için orijinal çizgede de böyle bir kenar olması gerekir. Bununla beraber eşleşen çizge bir kaynak ve musluk düğümü içerir ve kaynak düğümünden bütün sol düğümlere ve sağ düğümlerden musluğa kenar bulunur.

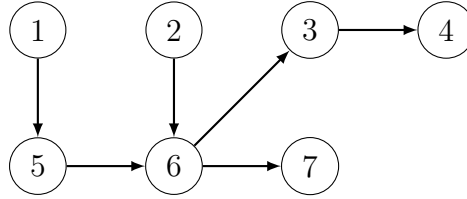
Sonuç çizgede bulunan maksimum eşleşme orijinal çizgedeki minimum düğüm ayırık yol kaplamasına karşılık gelir. Örneğin yukarıdaki çizge için aşağıdaki eşleşen çizgede 4 büyüklüğünde bir maksimum eşleşmesi içerir:



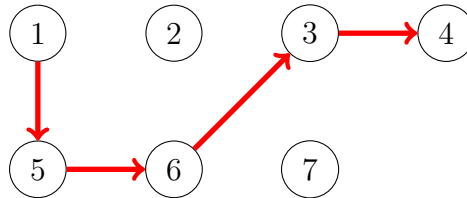
Eşleşen çizgedeki maksimum eşleşmedeki her kenar, orijinal çizgedeki kenar minimum ayırık düğüm yol kaplamasında bulunan kenara karşılık gelir. Bu yüzden minimum düğüm ayırık yol kaplamasının büyüklüğü $n - c$ olur. Burada n orijinal çizgedeki düğüm sayısını ve c ise maksimum eşleşmesinin büyüklüğünü gösterir.

Genel Yol Kaplaması

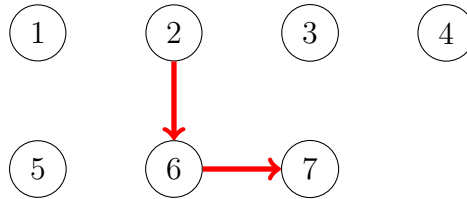
Genel yol kaplaması bir düğümün birden fazla yolda bulunabildiği yol kaplamasıdır. Minimum genel yol kaplaması minimum düğüm ayrık yol kaplamasından küçük olabilir çünkü bir düğüm farklı yollarda kullanılabilir. Aşağıdaki çizgeye bakalım:



Çizgenin minimum genel yol kaplaması iki yoldan oluşur. Örneğin, ilk yol aşağıdaki gibi olabilir:

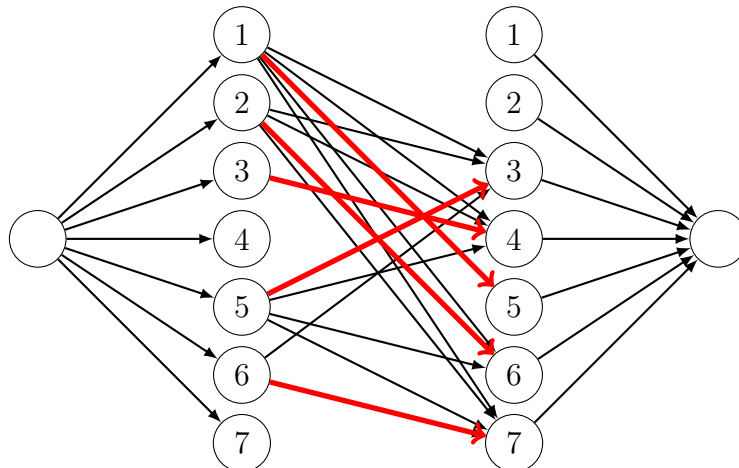


İkinci yol da aşağıdaki gibi olabilir:



Minimum genel yol kaplaması neredeyse minimum düğüm ayrık yol kaplaması gibi bulunabilir. a 'dan b 'ye orijinal çizgede yol olduğu zaman (birkaç kenardan geçen) eşleşen çizgede $a \rightarrow b$ kenarı bulunur. Bunu sağlamak için de eşleşen çizgeye kenar eklemek yeterlidir.

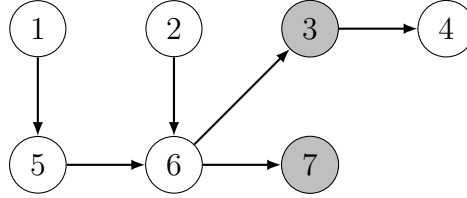
Yukarıdaki çizgenin eşleşen çizgesi aşağıdaki gibidir:



Dilworth'un Teoremi

Anti zincir çizgenin düğümlerinden oluşan bir grup olup herhangi bir düğümden diğer bir düğüme ulaşamamaktadır. **Dilworth'un Teoremi** yönlü asiklik bir çizgede bulunan minimum genel yol kaplaması maksimum anti zincirin büyüklüğüne eşit olduğunu gösterir.

Örneğin, yukarıdaki çizgede 3 ve 7. düğümler anti zincir oluşturur:



Bu bir maksimum anti zincirdir çünkü 3 düğüm içeren bir anti zincir oluşturmak mümkün değildir. Bunu daha önce çizgenin minimum genel yol kaplamasının 2 yoldan oluştuğunu görmüştük.

Kısım III

İleri Konular

Bölüm 21

Sayılar Teorisi

Sayılar teorisi matematiğin bir alt dalı olup tam sayılarla ilgilenir. Sayılar teorisi ilginç bir konudur çünkü tam sayı içeren çoğu soru başta kolay görünse de çözülmesi çok zordur.

Örneğin bu eşitliğe bakalım:

$$x^3 + y^3 + z^3 = 33$$

Çözümü sağlayan üç tane x, y, z reel sayıları bulmak kolaydır. Örneğin

$$\begin{aligned} x &= 3, \\ y &= \sqrt[3]{3}, \\ z &= \sqrt[3]{3}. \end{aligned}$$

Fakat bu eşitliği sağlayan üç *tam sayıyı* bulmak ise sayılar teorisinde açık bir problemdir [6].

Bu bölümde, sayılar teorisinde basit konseptler ile algoritmalara odaklanacağız. Bölüm içeriğinde aksi iddia edilmezse bütün sayılar tam sayılardır.

21.1 Asal sayılar(Primes) ve Bölenler(Factor)

Bir a sayısı b sayısını bölüyorsa a sayısına b sayısının **böleni** veya **çarpanı** denir. Eğer a, b sayısının böleni ise $a \mid b$ ve eğer değilse $a \nmid b$ yazılır. Örneğin 24'ün bölenleri 1, 2, 3, 4, 6, 8, 12 ve 24 olur.

Bir sayı $n > 1$ ve pozitif bölenleri sadece 1 ve n ise bu sayı bir **asal** sayıdır. Örneğin 7, 19 ve 41 asal sayılardır fakat 35 asal sayı değildir çünkü $5 \cdot 7 = 35$ idir. Her $n > 1$ sayısı için farklı bir asal çarpanlara ayırma

$$n = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_k^{\alpha_k},$$

vardır. Burada p_1, p_2, \dots, p_k farklı asal sayılar olur ve $\alpha_1, \alpha_2, \dots, \alpha_k$ pozitif sayılardır. Örneğin, 84'ün asal çarpanlara ayrılmış hali

$$84 = 2^2 \cdot 3^1 \cdot 7^1$$

olur.

n sayısının **çarpan sayısı**

$$\tau(n) = \prod_{i=1}^k (\alpha_i + 1),$$

olur çünkü her p_i asal sayısı için onun kaç defa kullanılmasının $\alpha_i + 1$ yolu vardır. Örneğin 84'ün çarpanlarının sayısı $\tau(84) = 3 \cdot 2 \cdot 2 = 12$. 84'ün çarpanları 1, 2, 3, 4, 6, 7, 12, 14, 21, 28, 42 ve 84 olur.

n sayısının **çarpanların toplamı**

$$\sigma(n) = \prod_{i=1}^k (1 + p_i + \dots + p_i^{\alpha_i}) = \prod_{i=1}^k \frac{p_i^{\alpha_i+1} - 1}{p_i - 1},$$

idir ki ikinci formül geometrik dizi formülüdür. Örneğin 84'ün katsayılar toplamı

$$\sigma(84) = \frac{2^3 - 1}{2 - 1} \cdot \frac{3^2 - 1}{3 - 1} \cdot \frac{7^2 - 1}{7 - 1} = 7 \cdot 4 \cdot 8 = 224.$$

n sayısının **katsayılar çarpımı**

$$\mu(n) = n^{\tau(n)/2},$$

çünkü çarpanlardan $\tau(n)/2$ çift oluşturabiliriz. Örneğin 84 sayısının katsayıları $1 \cdot 84$, $2 \cdot 42$, $3 \cdot 28$ çiftlerini oluşturur ve bu çarpanların çarpımları $\mu(84) = 84^6 = 351298031616$ olur.

Bir n sayısı $n = \sigma(n) - n$ ise mükemmel sayıdır (perfect number) yani n , 1'den $n - 1$ 'e kadar olan çarpanların toplamına eşittir. Örneğin 28 mükemmel sayıdır çünkü $28 = 1 + 2 + 4 + 7 + 14$ olur.

Asal Sayıların Sayısı

Sonsuz sayıda asal sayı bulunduğunu göstermek kolaydır. Eğer asal sayı sayısı sınırlı olsaydı $P = \{p_1, p_2, \dots, p_n\}$ şeklinde bütün asalları içeren bir grup oluşturabiliriz. Örneğin $p_1 = 2$, $p_2 = 3$, $p_3 = 5$ diye gider. Fakat P 'yi kullanarak P 'deki bütün elemanlardan büyük bir

$$p_1 p_2 \cdots p_n + 1$$

asal sayısı oluşturulabilir. Bu da bir çelişkidir ve asal sayı sayısı sonsuzdur.

Asal Sayıların Yoğunluğu

Asal sayıların yoğunluğu, sayılar arasında ne sıklıkla asal sayı olduğunu ifade eder. Diyelim ki $\pi(n)$ 1'den n 'e kadar olan asal sayıların sayısını belirtsin. Örneğin $\pi(10) = 4$ olur çünkü 1 ve 10 arasında 4 asal sayı vardır: 2, 3, 5 ve 7 olur.

$$\pi(n) \approx \frac{n}{\ln n},$$

diye göstermek mümkündür yani asal sayılar çok sıklıkla bulunur. Örnek olarak 1 ve 10^6 arasındaki asal sayılar $\pi(10^6) = 78498$ olur ve $10^6 / \ln 10^6 \approx 72382$ idir.

Konjektürler (Conjectures)

Asal sayıları içeren bir sürü *konjektür* vardır. Çoğu insan konjektürlerin doğru olduğunu düşünür fakat hiç kimse onları kanıtlayamamıştır. Örneğin aşağıdaki konjektürler ünlüdür:

- **Goldbach'ın Konjektürü:** Her çift $n > 2$ sayısı a ve b 'nin asal sayı olacak şekilde $n = a + b$ şeklinde gösterilebilir.
- **İkiz Asal Sayı Konjektürü:** p ve $p + 2$ sayılarının ikisinin de asal sayı olup $\{p, p + 2\}$ şeklinde ifade edilebilecek sonsuz sayıda çift vardır.
- **Legendre'nin Konjektürü:** n pozitif sayıysa n^2 ile $(n + 1)^2$ sayıları kesin bir asal sayı vardır.

Basit Algoritmaları

Eğer n asal sayı değilse $a \leq \sqrt{n}$ veya $b \leq \sqrt{n}$ olduğu zaman $a \cdot b$ çarpımı şeklinde gösterilebilir. Yani 2 ve $\lfloor \sqrt{n} \rfloor$ arasında kesinlikle bir çarpan vardır. Bu gözlemi kullanarak bir sayının asal sayı olup olmadığını ve sayının asal çarpanlarına ayrılmasını $O(\sqrt{n})$ zamanda bulunur.

prime fonksiyonu n sayısının asal sayı olup olmadığını bulur. Bu fonksiyon n 'i 2 den $\lfloor \sqrt{n} \rfloor$ kadar olan bütün sayılara bölmeyi çalışır. Eğer hiçbiri n 'i bölemiyorsa n asal sayıdır.

```
bool prime(int n) {
    if (n < 2) return false;
    for (int x = 2; x*x <= n; x++) {
        if (n%x == 0) return false;
    }
    return true;
}
```

factors fonksiyonu, n sayısının asal çarpanlarını içerir. Fonksiyon n sayısını asal çarpanlarına böler ve onları vektöre ekler. Fonksiyon kalan n sayısı 2 and $\lfloor \sqrt{n} \rfloor$ sayıları arasında çarpanı kalmayınca bitiyor. Eğer $n > 1$ ise asal sayıdır ve son çarpanıdır.

```
vector<int> factors(int n) {
    vector<int> f;
    for (int x = 2; x*x <= n; x++) {
        while (n%x == 0) {
            f.push_back(x);
            n /= x;
        }
    }
    if (n > 1) f.push_back(n);
    return f;
}
```

Sayının her asal çarpanı kaç defa bulunuyorsa vektörde o kadar miktarda bulunur. Örneğin $24 = 2^3 \cdot 3$ idir ve fonksiyon sonuç olarak $[2, 2, 2, 3]$ verir.

Eratosten Kalburu (Sieve of Eratosthenes)

Eratosten kalburu $2 \dots n$ arasında verilen bir sayının asal sayı olup olmadığını ve eğer değilse sayının bir asal çarpanını bulmasını sağlayan bir ön işleme algoritmasıdır. Bu algoritma dizi oluşturarak verimli bir şekilde çalışır.

Algoritma *sieve* dizisi oluşturur ve $2, 3, \dots, n$ pozisyonları kullanılır. $\text{sieve}[k] = 0$ k sayısının asal sayı olduğunu ve $\text{sieve}[k] \neq 0$ ise k sayısı asal sayı değildir ve sayının asal sayılarından biri $\text{sieve}[k]$ olur.

Algoritma $2 \dots n$ sayılarından teker teker geçer ve eğer yeni bir x asal sayısı bulunursa algoritma x 'in katlarını $(2x, 3x, 4x, \dots)$ asal sayı değil diyerek işaretlenir çünkü x sayısı bunları böler.

Örneğin, $n = 20$ ise dizi şekildeki gibidir:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	2	0	3	0	2	3	5	0	3	0	7	5	2	0	3	0	5

Aşağıdaki kod, Eratosten kalburunun çalıştırır. Kod, *sieve* dizisindeki her elemanın başta 0 olduğunu varsayar.

```
for (int x = 2; x <= n; x++) {
    if (sieve[x]) continue;
    for (int u = 2*x; u <= n; u += x) {
        sieve[u] = x;
    }
}
```

Algoritmanın iç döngüsü her x değeri için n/x defa çalışır. Böylece algoritmanın çalışma zamanının üst sınırı

$$\sum_{x=2}^n n/x = n/2 + n/3 + n/4 + \dots + n/n = O(n \log n).$$

harmonik toplamı olur.

Aslında algoritma daha verimlidir çünkü iç döngü sadece x asal sayı olursa çalışır. Algoritmanın çalışma zamanının sadece $O(n \log \log n)$ olduğu gösterilebilir ve bu $O(n)$ zaman karmaşıklığına çok yakındır.

Öklid'in Algoritması (Euclid's Algorithm)

a ve b sayılarının **en büyük ortak böleni**, $\text{gcd}(a, b)$, hem a hem de b 'yi bölen en büyük sayıdır ve a ile b sayılarının **en küçük ortak katı**, $\text{lcm}(a, b)$, ise hem a hem de b tarafından bölünebilen en küçük sayıdır. Örneğin $\text{gcd}(24, 36) = 12$ ve $\text{lcm}(24, 36) = 72$ olur.

En büyük ortak bölen ile en küçük ortak katı aşağıdaki gibi birbirlerine bağlıdır:

$$\text{lcm}(a, b) = \frac{ab}{\text{gcd}(a, b)}$$

Öklid'in Algoritması¹ iki sayının en büyük ortak bölenini bulmayı sağlayan verimli bir algoritmadır. Algoritma aşağıdaki formüle dayanır:

$$\text{gcd}(a, b) = \begin{cases} a & b = 0 \\ \text{gcd}(b, a \bmod b) & b \neq 0 \end{cases}$$

Örneğin

$$\text{gcd}(24, 36) = \text{gcd}(36, 24) = \text{gcd}(24, 12) = \text{gcd}(12, 0) = 12.$$

Algoritma aşağıdaki gibi koda geçirilebilir:

```
int gcd(int a, int b) {
    if (b == 0) return a;
    return gcd(b, a%b);
}
```

Öklid'in Algoritmasının $n = \min(a, b)$ olduğu yerde $O(\log n)$ sürede çalıştığını göstermek mümkündür. Algoritmanın en kötü durumunda a ve b ardışık Fibonacci sayılarıdır. Örneğin

$$\text{gcd}(13, 8) = \text{gcd}(8, 5) = \text{gcd}(5, 3) = \text{gcd}(3, 2) = \text{gcd}(2, 1) = \text{gcd}(1, 0) = 1.$$

Euler'in Totient Fonksiyonu

Eğer $\text{gcd}(a, b) = 1$ ise a ve b sayıları aralarında asaldır (coprime). **Euler'in Totient fonksiyonu** $\varphi(n)$ n sayısının 1'den n kadar olan aralarında asal sayı sayısını verir. Örneğin $\varphi(12) = 4$ çünkü 1, 5, 7 ve 11 sayıları 12'yle aralarında asaldır.

$\varphi(n)$ değeri n sayısının asal çarpanlarına ayrılması ile

$$\varphi(n) = \prod_{i=1}^k p_i^{\alpha_i-1} (p_i - 1)$$

formülünden bulunabilir. Örneğin $\varphi(12) = 2^1 \cdot (2 - 1) \cdot 3^0 \cdot (3 - 1) = 4$. Bu arada n asal sayı ise $\varphi(n) = n - 1$ olur.

21.2 Modüler Aritmetik

Modüler aritmetikte kullanılan sayılar m 'in sabit bir sayı olduğu kabul edilerek $0, 1, 2, \dots, m-1$ ile sınırlıdır. Her x sayısı $x \bmod m$ ile gösterilir yani x sayısını

¹Öklid M.Ö. 300'lerde yaşayan bir Yunan matematikçidir. Belki de bu tarihte bilinen ilk algoritmadır.

m sayısına bölünce kalan ile gösterilir. Örneğin $m = 17$ ise 75 sayısı $75 \bmod 17 = 7$ ile gösterilir.

Genelde hesaplama yapmadan önce kalanları alabiliriz. Spesifik olarak aşağıdaki formüller kullanılır:

$$\begin{aligned}(x + y) \bmod m &= (x \bmod m + y \bmod m) \bmod m \\(x - y) \bmod m &= (x \bmod m - y \bmod m) \bmod m \\(x \cdot y) \bmod m &= (x \bmod m \cdot y \bmod m) \bmod m \\x^n \bmod m &= (x \bmod m)^n \bmod m\end{aligned}$$

Modüler Üs Alma (Modular Exponentiation)

Genelde verimli bir şekilde $x^n \bmod m$ değerini hesaplamak gerekir. Bu $O(\log n)$ zamanda aşağıdaki özyineleme ile yapılabilir:

$$x^n = \begin{cases} 1 & n = 0 \\ x^{n/2} \cdot x^{n/2} & n \text{ çift} \\ x^{n-1} \cdot x & n \text{ tek} \end{cases}$$

n , çift sayısı olduğu zaman $x^{(n/2)}$ değeri sadece bir defa hesaplanır. Bu algoritmanın zaman karmaşıklığının $O(\log n)$ olmasını garantiler çünkü n çift olduğu zaman her zaman yarıya iner.

Aşağıdaki fonksiyon $x^n \bmod m$ değerini hesaplar:

```
int modpow(int x, int n, int m) {
    if (n == 0) return 1%m;
    long long u = modpow(x, n/2, m);
    u = (u*u)%m;
    if (n%2 == 1) u = (u*x)%m;
    return u;
}
```

Fermat'ın Teoremi ve Euler'in Teoremi

Fermat'ın Teoremi m asal sayı ve x ile m aralarında asal olduğu zaman

$$x^{m-1} \bmod m = 1$$

olduğunu söyler. Bu aynı zamanda

$$x^k \bmod m = x^{k \bmod (m-1)} \bmod m$$

verir. Genel olarak **Euler'in Teoremi** x ile m aralarında asal olduğu

$$x^{\varphi(m)} \bmod m = 1$$

olur. Fermat'ın Teoremi, Euler'in Teoremi'nin devamı gibidir çünkü m asal sayı ise $\varphi(m) = m - 1$ olur.

Modüler Çarpımsal Ters Bulma (Modular Inverse)

x modulo m ifadesinin tersi x^{-1} olur ki bu

$$xx^{-1} \bmod m = 1.$$

Örneğin $x = 6$ ve $m = 17$ ise $x^{-1} = 3$ olur çünkü $6 \cdot 3 \bmod 17 = 1$.

Modüler çarpımsal ters bulmayı kullanarak sayı modulo m sayılarını bölebiliriz çünkü x 'e bölmek x^{-1} sayısı ile çarpmaya eşdeğerdir. Örneğin $36/6 \bmod 17$ değerini hesaplamak için $2 \cdot 3 \bmod 17$ formülünü kullanarak $2 \cdot 3 \bmod 17$ formülünü kullanabiliriz çünkü $36 \bmod 17 = 2$ ve $6^{-1} \bmod 17 = 3$ olur.

Fakat, bir modüler çarpımsal ters her zaman bulunmaz. Örneğin $x = 2$ ve $m = 4$ olursa

$$xx^{-1} \bmod m = 1$$

eşitliği çözülemez çünkü 2'nin her katı çifttir ve $m = 4$ olduğu zaman kalan asla 1 olamaz. $x^{-1} \bmod m$ değeri kesin olarak x ve m sayıları aralarında asal olduğu zaman bulunabilir.

Eğer modüler çarpımsal ters varsa o zaman aşağıdaki formül kullanılarak bulunabilir:

$$x^{-1} = x^{\varphi(m)-1}.$$

Eğer m asal sayıysa, formül

$$x^{-1} = x^{m-2}.$$

olur. Örneğin,

$$6^{-1} \bmod 17 = 6^{17-2} \bmod 17 = 3.$$

Bu formül bize modüler üs alma algoritmasını kullanarak modüler çarpımsal tersini verimli bir şekilde bulmamızı sağlar. Bu formül Euler'in teoremi kullanılarak türetilir. İlk başta, modüler çarpımsal ters aşağıdaki eşitliği sağlamalıdır:

$$xx^{-1} \bmod m = 1.$$

Diğer tarafta Euler'in teoremine göre

$$x^{\varphi(m)} \bmod m = xx^{\varphi(m)-1} \bmod m = 1,$$

yani x^{-1} ve $x^{\varphi(m)-1}$ eşit olur.

Bilgisayar Aritmetiği

Programlamada, unsigned tam sayılar 2^k ile gösterilir ve k ise veri birimindeki bit sayısını gösterir. Bunun sonucu olarak bir sayı çok büyükse sarılır.

Örneğin C++'da unsigned tam sayılar 2^{32} ile gösterilir. Aşağıdaki kod değeri 123456789 olan bir unsigned tam sayı tanımlar. Bundan sonra değer kendisiyle çarpılır ve sonuç olarak $123456789^2 \bmod 2^{32} = 2537071545$ olur.

```
unsigned int x = 123456789;
cout << x*x << "\n"; // 2537071545
```

21.3 Eşitlikleri Çözmek

Diyofantus Eşitlikleri

Diyofantus eşitliği a , b ve c 'lerin sabit sayı olduğu varsayılarak

$$ax + by = c$$

şeklinde eşitliktir. Burada x ve y değerleri bulunur. Eşitlikte her sayı bir tam sayı olmalıdır. Örneğin $5x + 2y = 11$ eşitliği için bir çözüm $x = 3$ ve $y = -2$ olur.

Diyofantus eşitliği çözmek için Öklid'in algoritması kullanılır. Öklid'in algoritmasını genişleterek eşitliği sağlayacak x ve y sayıları bulabiliriz:

$$ax + by = \gcd(a, b)$$

Diyofantus eşitliği c $\gcd(a, b)$ 'e bölünebiliyorsa çözülebilir aksi taktirde çözülemez.

Örneğin, eşitliği sağlayan x ve y sayılarını bulalım:

$$39x + 15y = 12$$

Eşitlik çözülebilir çünkü $\gcd(39, 15) = 3$ ve $3 \mid 12$. Öklid'in algoritması 39 ile 15'in en büyük ortak bölenlerini hesaplarken aşağıdaki fonksiyon çağrılarını oluşturur:

$$\gcd(39, 15) = \gcd(15, 9) = \gcd(9, 6) = \gcd(6, 3) = \gcd(3, 0) = 3$$

Bu aşağıdaki eşitliklere karşılık gelir:

$$\begin{aligned} 39 - 2 \cdot 15 &= 9 \\ 15 - 1 \cdot 9 &= 6 \\ 9 - 1 \cdot 6 &= 3 \end{aligned}$$

Bu eşitlikleri kullanarak

$$39 \cdot 2 + 15 \cdot (-5) = 3$$

ifadesini türetebiliriz ve 4'le çarparak sonuç

$$39 \cdot 8 + 15 \cdot (-20) = 12,$$

yani eşitlik bir çözüm $x = 8$ ve $y = -20$ olur.

Diyofantus eşitliğinin tek bir çözümü yoktur çünkü bir çözümü biliyorsak sonsuz tane çözüm oluşturabiliriz. Eğer (x, y) çifti bir çözümse

$$\left(x + \frac{kb}{\gcd(a, b)}, y - \frac{ka}{\gcd(a, b)}\right)$$

olan k 'nin herhangi bir tam sayı olan tüm çiftler de çözülebilir.

Çin Kalan Teoremi (Chinese Remainder Theorem)

Çin Kalan Teoremi aşağıdaki formda olan eşitlik gruplarını çözer:

$$\begin{aligned}x &= a_1 \bmod m_1 \\x &= a_2 \bmod m_2 \\&\dots \\x &= a_n \bmod m_n\end{aligned}$$

bütün m_1, m_2, \dots, m_n çiftleri aralarında asaldır.

x_m^{-1} x modulo m 'in tersidir ve

$$X_k = \frac{m_1 m_2 \cdots m_n}{m_k}$$

olur Bu gösterimi kullanarak eşitliklerin bir çözümü

$$x = a_1 X_1 X_{1m_1}^{-1} + a_2 X_2 X_{2m_2}^{-1} + \cdots + a_n X_n X_{nm_n}^{-1}$$

olur. Bu çözümde, her $k = 1, 2, \dots, n$ için

$$a_k X_k X_{km_k}^{-1} \bmod m_k = a_k,$$

çünkü

$$X_k X_{km_k}^{-1} \bmod m_k = 1.$$

Toplamdaki diğer bütün terimler m_k 'ye bölünebildiği için kalanda etkileri yoktur ve $x \bmod m_k = a_k$ olur.

Örneğin

$$\begin{aligned}x &= 3 \bmod 5 \\x &= 4 \bmod 7 \\x &= 2 \bmod 3\end{aligned}$$

çözümü

$$3 \cdot 21 \cdot 1 + 4 \cdot 15 \cdot 1 + 2 \cdot 35 \cdot 2 = 263$$

olur.

Bir x çözümü bulduğumuz zaman sonsuz sayıda başka çözüm bulmuş oluruz çünkü bütün

$$x + m_1 m_2 \cdots m_n$$

formundaki sayılar çözümdür.

21.4 Diğer Sonuçlar

Lagrange'in Teoremi

Lagrange'in teoremi her pozitif tam sayının dört tane karenin toplamı olarak gösterilebileceğini söyler yani $a^2 + b^2 + c^2 + d^2$ olur. Örneğin 123 sayısı $8^2 + 5^2 + 5^2 + 3^2$ şeklinde gösterilebilir.

Zeckendorf'in Teoremi

Zeckendorf'in teoremi her pozitif tam sayının farklı bir Fibonacci sayı toplamı şeklinde gösterimi vardır. Örneğin 74 sayısı $55 + 13 + 5 + 1$ şeklinde gösterilebilir.

Pisagor Üçlüleri

Pisagor üçlüsü (a, b, c) şeklinde olup $a^2 + b^2 = c^2$ Pisagor teoremini sağlar yani bunlar kenar uzunlukları a, b, c olan bir dik üçgen oluşturur. Örneğin $(3, 4, 5)$ bir Pisagor üçlüsüdür.

Eğer (a, b, c) bir Pisagor üçlüsü ise bütün $k > 1$ 'nin olduğu bütün (ka, kb, kc) formundaki üçlüler de Pisagor üçlüsüdür. Eğer a, b ve c sayıları aralarında asalsa bunların oluşturacağı Pisagor üçlüsü *basittir* (primitive). Bütün Pisagor üçlüleri bir k katsayısı kullanarak bu basit üçlülerden oluşturulabilir.

Öklid'in formülü bütün basit Pisagor üçlüsü oluşturması için kullanılabilir. Bu tip üçlünün formu

$$(n^2 - m^2, 2nm, n^2 + m^2),$$

şeklindedir. Burada $0 < m < n$, n ve m aralarında asal olup en az n ve m 'den birisi çift olmalıdır. Örneğin $m = 1$ ve $n = 2$ olduğu durumda formül en küçük Pisagor üçlüsü olarak

$$(2^2 - 1^2, 2 \cdot 2 \cdot 1, 2^2 + 1^2) = (3, 4, 5)$$

oluşturur.

Wilson'un Teoremi

Wilson'un Teoremi n sayısının

$$(n - 1)! \bmod n = n - 1$$

olduğu zaman asal sayı olduğunu söyler. Örneğin 11 asal sayıdır çünkü

$$10! \bmod 11 = 10$$

ve 12 sayısı asal sayı değildir çünkü

$$11! \bmod 12 = 0 \neq 11$$

olur.

Böylece Wilson'un Teoremi kullanılarak bir sayının asal olup olmadığı bulunabilir. Fakat, pratikte bu teorem büyük n sayılarına uygulanamaz çünkü n büyük olduğu zaman $(n - 1)!$ değerini hesaplamak zordur.

Bölüm 22

Kombinatorik (Combinatorics)

Kombinatorik objelerin kombinasyonlarını saymamızı sağlayan yöntemleri araştırır. Genelde amaç, her kombinasyonu ayrı ayrı oluşturmadan toplam kombinasyon sayısını hesaplamaktır.

Örneğin toplamları n tam sayısını oluşturan tam sayı kombinasyonlarını bulmak olsun. Örneğin 4 için 8 durum var:

- $1 + 1 + 1 + 1$
- $1 + 1 + 2$
- $1 + 2 + 1$
- $2 + 1 + 1$
- $2 + 2$
- $3 + 1$
- $1 + 3$
- 4

Bir kombinasyon sorusu özyinelemeli fonksiyon ile çözülebilir. Bu problemde n sayısı için toplam gösterim sayısını veren bir $f(n)$ fonksiyonu tanımlayacağız. Örneğin yukarıdaki örneğe göre $f(4) = 8$ olur. Fonksiyonun değerleri özyinelemeli bir şekilde aşağıdaki gibi hesaplanabilir:

$$f(n) = \begin{cases} 1 & n = 0 \\ f(0) + f(1) + \dots + f(n-1) & n > 0 \end{cases}$$

Temel durum $f(0) = 1$ olur çünkü boş toplam 0 sayısını temsil eder. Sonrasında eğer $n > 0$ ise toplamın ilk sayısını bulmak için bütün yolları buluruz. Eğer ilk sayı k ise toplamın geri kalan kısmı $f(n-k)$ tan gösterimle gösterilebilir. Böylece $k < n$ olduğu durumda bütün $f(n-k)$ değerlerinin toplamını hesaplarız.

Fonksiyonun ilk değerleri:

$$\begin{aligned} f(0) &= 1 \\ f(1) &= 1 \\ f(2) &= 2 \\ f(3) &= 4 \\ f(4) &= 8 \end{aligned}$$

Baszen özyinelemeli bir formül bir kapalı formdaki bir formülle değiştirilebilir. Örneği bu problemde

$$f(n) = 2^{n-1},$$

bu formül + ve - işaretleri için $n - 1$ tane uygun yer olmasından dolayı gelir ve istediğimiz alt kümeyi seçebiliriz.

22.1 Binom Katsayıları (Binomial Coefficients)

Binom katsayısı $\binom{n}{k}$ n tane elemandan oluşan bir kümede k elemandan oluşan kaç tane alt küme seçebileceğimize eşittir. Örneğin $\binom{5}{3} = 10$ çünkü $\{1, 2, 3, 4, 5\}$ kümesinin 3 elemandan oluşan 10 tane alt kümesi vardır:

$\{1, 2, 3\}, \{1, 2, 4\}, \{1, 2, 5\}, \{1, 3, 4\}, \{1, 3, 5\}, \{1, 4, 5\}, \{2, 3, 4\}, \{2, 3, 5\}, \{2, 4, 5\}, \{3, 4, 5\}$

1. Formül

Binom katsayıları özyinelemeli bir şekilde aşağıdaki gibi hesaplanabilir:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Buradaki fikir kümedeki x elemanını sabitlemektir. Eğer x alt kümede bulunuyorsa $n - 1$ elemandan $k - 1$ tane eleman seçmemiz gerekir ve eğer x alt kümede bulunmuyorsa $n - 1$ elemandan k tane eleman seçmemiz gerekir.

Özyinelemedeki temel durumlar

$$\binom{n}{0} = \binom{n}{n} = 1,$$

çünkü boş bir küme ve bütün elemanları içeren bir küme oluşturmanın sadece tek bir yolu vardır.

2. Formül

Binom katsayılarını hesaplamanın başka bir yolu aşağıdaki gibidir:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

n tane elemanın $n!$ tane permütasyonu vardır. Bütün permütasyonlardan geçeriz ve her seferinde permütasyondaki ilk k elemanı alt kümeye ekleriz. Alt kümenin içindeki ve dışındaki elemanların sıralaması önemli olmadığı için sonuç $k!$ ve $(n - k)!$ ifadelerine bölünür.

Özellikler

Binom katsayıları için

$$\binom{n}{k} = \binom{n}{n-k},$$

çünkü n elemandan oluşan bir kümeyi iki alt kümeye böleriz: İlki k eleman içerirken ikincisi $n - k$ eleman içerir.

Binom katsayılarının toplamı ise

$$\binom{n}{0} + \binom{n}{1} + \binom{n}{2} + \dots + \binom{n}{n} = 2^n.$$

Bu şeyin isminin "binom katsayısı" olmasının nedeni ise $(a + b)$ binomunun n . kuvvetinde görülebilir:

$$(a + b)^n = \binom{n}{0}a^n b^0 + \binom{n}{1}a^{n-1}b^1 + \dots + \binom{n}{n-1}a^1b^{n-1} + \binom{n}{n}a^0b^n.$$

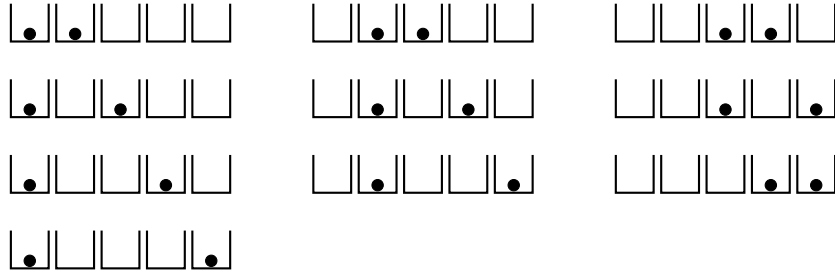
Binom katsayıları aynı zamanda **Pascal'ın üçgeninde** de görülebilir. Buradaki her değer yukarıdaki iki değerinin toplamıdır:

$$\begin{array}{ccccccccc} & & & & 1 & & & & \\ & & & 1 & & 1 & & & \\ & & 1 & & 2 & & 1 & & \\ & 1 & & 3 & & 3 & & 1 & \\ 1 & & 4 & & 6 & & 4 & & 1 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \end{array}$$

Kutular ve Toplar

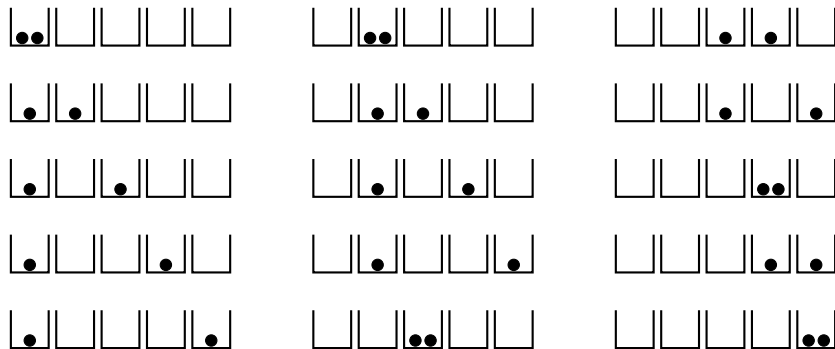
"Kutular ve top" k tane topu n tane kutuya kaç defa yerleştirebileceğimizi bulduğumuz işe yarar bir modeldir. Üç olası durumu düşünelim:

1. *Durum:* Her kutu en fazla 1 top barındırabilir. Örneğin $n = 5$ ve $k = 2$ ise 10 çözüm vardır:



Bu durumda çözüm direkt $\binom{n}{k}$ binom katsayısı olur.

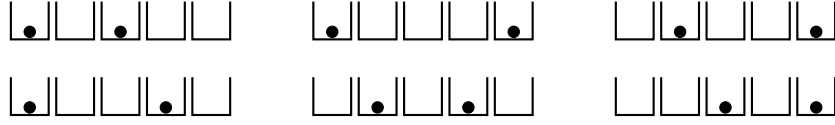
2. *Durum:* Bir kutu birkaç tane top içerebilir. Örneğin $n = 5$ ve $k = 2$ olduğu zaman 15 tane çözüm vardır:



Burada topları kutuya yerleştirme işlemi "o" ve "→" sembollerinden oluşan bir yazıyla(string) gösterilebilir. Başta en soldaki kutuda durduğumuz varsayılar ve "o" sembolü şu anki kutuya bir top yerleştirdiğimiz anlamına ve "→" ise bir sağdaki kutuya gittiğimiz söyler.

Bu gösterimi kullanarak her çözüm k defa "o" sembolünden ve $n - 1$ tane de "→" sembolünden oluşan bir yazı olur. Örneğin yukarıdaki resimdeki sağ üst çözüm "→ → o → o →" yazısına denk gelir. Böylece toplam çözüm sayısı $\binom{k+n-1}{k}$ olur.

3. *Durum*: Her kutu en fazla bir top içerir ve hiçbir ardışık iki kutu aynı anda top içeremez. Örneğin $n = 5$ ve $k = 2$ olduğu zaman 6 çözüm vardır:



Bu durumda k topun ilk başta kutulara yerleştirilip her ardışık iki kutu arasında bir boş kutu olduğunu varsayabiliriz. Şimdiki amacımız kalan boş kutular için uygun yerler seçmektir. Bu tip $n - 2k + 1$ kutu ve $k + 1$ tane yer vardır. Böylece 2. senaryodaki formülü kullanarak toplam çözüm sayısını $\binom{n-k+1}{n-2k+1}$ bulabiliriz.

Çok Terimli Katsayılar (Multinomial coefficients)

Çok terimli katsayılar

$$\binom{n}{k_1, k_2, \dots, k_m} = \frac{n!}{k_1! k_2! \dots k_m!},$$

n tane elemanı k_1, k_2, \dots, k_m büyüklüğündeki alt kümelerine ayırma sayısını verir. Çok terimli katsayılar binom katsayıların genellemesi gibi görülebilir. Eğer $m = 2$ ise yukarıdaki formül binom katsayı formülüne denk gelir.

22.2 Katalan Sayıları

Katılan sayısı C_n , n tane sol parantez ve n tane sağ parantezden oluşan uygun parantez gösteriminin toplam sayısıdır.

Örneğin $C_3 = 5$ çünkü 3 tane sol ve sağ parantezleri kullanarak aşağıdaki parantez ifadelerini oluşturabiliriz:

- $()()()$
- $((()))$
- $()(())$
- $((())())$
- $((()()))$

Parentez Gösterimi

Uygun parentez gösterimi tam olarak ne oluyor? Aşağıdaki kurallar bütün uygun parentez gösterimlerini tanımlar:

- Boş parentez gösterimi uygundur.
- Eğer A gösterimi uygunsa (A) gösterimi de uygundur.
- Eğer A ve B gösterimleri uygunsa AB gösterimi de uygundur.

Uygun parentez gösterimlerini karakterize etmenin başka bir yolu ise eğer herhangi bir gösteriminin ön kısmını alırsak bu kısım en az içerdği sağ parantez kadar sol parantez içermelidir (daha fazla sol parantez içerebilir.). Bununla beraber tam gösterim eşit sayıda sol ve sağ parantez sayısı içermelidir.

1. Formül

Katalan sayıları aşağıdaki formül kullanılarak hesaplanabilir:

$$C_n = \sum_{i=0}^{n-1} C_i C_{n-i-1}.$$

Toplam, ifadeyi iki parçaya bölebilecek yollardan geçer ve bu yollarda her iki parçanın da uygun gösterim olmasına ve ilk parçanın boş olmayacak şekilde olabildiğince küçük olmasına dikkat eder. Herhangi bir i için ilk bölüm $i + 1$ parantez çifti içerir ve toplam gösterim sayısı aşağıdaki değerlerin çarpımıdır:

- C_i : en dıştaki parantezleri saymadan ilk bölümdeki parantezleri kullanarak oluşturulabilecek gösterim sayısı
- C_{n-i-1} : ikinci bölümdeki parantezleri kullanarak oluşturulabilecek gösterim sayısı

Temel durum $C_0 = 1$ olur çünkü sıfır parantez çiftinden boş bir parantez gösterimi oluşturabiliriz.

2. Formül

Katalan sayıları ayrıca binom katsayıları kullanılarak hesaplanabilir:

$$C_n = \frac{1}{n+1} \binom{2n}{n}$$

Formül şöyle açıklanabilir:

Uygun olup olmamasına dikkat etmeden n tane sol ve n tane sağ parantezden oluşan $\binom{2n}{n}$ tane parantez gösterimi oluşturabiliriz. Şimdi uygun *olmayan* gösterim sayısını bulalım.

Eğer bir parantez gösterimi uygun değilse, sağ parantez sayısının sol parantez sayısından daha büyük olduğu bir ön kısım içermelidir. Buradaki fikir bu ön

kısma ait olan bütün parentezleri tersin çevirmektir. Örneğin $()())()$ ifadesi $()()$ ön kısmı içerir ve ön kısmı terse çevirdikten sonra gösterim $)(((()$ olur.

Sonuç olarak gösterim $n + 1$ tane sol ve $n - 1$ sağ parentezden oluşur. Bu durumla beraber toplam gösterim sayısı $\binom{2n}{n+1}$ olur ve bu da toplam uygun olmayan gösterim sayısını verir. Böylece uygun parentez gösterim sayısı

$$\binom{2n}{n} - \binom{2n}{n+1} = \binom{2n}{n} - \frac{n}{n+1} \binom{2n}{n} = \frac{1}{n+1} \binom{2n}{n}$$

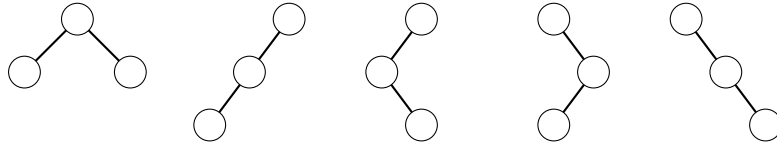
formülü kullanılarak hesaplanabilir.

Ağaçları Saymak

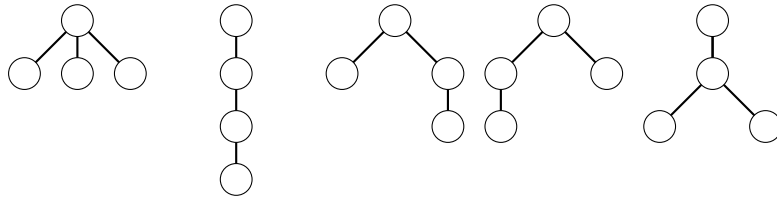
Katalan sayıları aynı zamanda ağaçlarla da ilgilidir:

- C_n tane n düğümden oluşan ikili ağaç vardır
- C_{n-1} tane n düğümden oluşan köklü ağaç vardır

Örneğin $C_3 = 5$ için ikili ağaçlar



köklü ağaçlar da



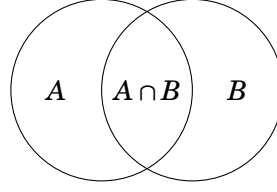
şeklinde olur.

22.3 İçerme-Dışarma (Inclusion-Exclusion)

İçerme-dışarma bir küme grubunun büyüklüğünü kesişim kısımlarının büyüklüğünü bildiğimizde veya tam tersi durumda bulmamıza yarar. Bu tekniği basit bir örneği

$$|A \cup B| = |A| + |B| - |A \cap B|,$$

formülü olur ve burada A ve B kümeysen $|X|$ ise X 'in büyüklüğünü verir. Formül aşağıdaki gibi gösterilebilir:

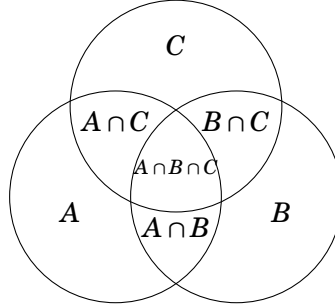


Buradaki amacımız $A \cup B$ grubunun yani en az bir dairenin içinde bulunan bölgenin büyüklüğünü bulmaktır. Resim bize $A \cup B$ alanını ilk başta A ve B alanlarını toplayıp sonra $A \cap B$ alanını çıkararak bulabileceğimizi gösterir.

Aynı fikir grup sayısı daha fazla olduğunda da uygulanabilir. 3 grup olduğu zaman, içerme-dışarma formülü

$$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|$$

olur ve buna karşılık gelen resim



Genel durumda, grubun büyüklüğü $X_1 \cup X_2 \cup \dots \cup X_n$, X_1, X_2, \dots, X_n gruplarından bazılarını içeren bütün kesişim kısımlarından geçip bulunabilir. Eğer kesişim tek sayıda grup içeriyorsa büyüklüğü sonuca eklenir ve eğer çift sayıda grup içeriyorsa büyüklüğü sonuçtan çıkarılır.

Grupların büyüklüklerinden kesişim noktalarını hesaplamamanın benzer formülleri vardır. Örneğin

$$|A \cap B| = |A| + |B| - |A \cup B|$$

ve

$$|A \cap B \cap C| = |A| + |B| + |C| - |A \cup B| - |A \cup C| - |B \cup C| + |A \cup B \cup C|$$

olur.

Düzensizlikler (Derangements)

Örnek olarak $\{1, 2, \dots, n\}$ elemanlarının **düzensizliklerini** sayalım yani hiçbir elemanın orijinal yerinde olmadığı permütasyonları bulalım. Örneğin $n = 3$ iken $(2, 3, 1)$ ve $(3, 1, 2)$ olmak üzere iki tane düzensizlik vardır.

Problemi çözmenin yollarından biri içerme-dışarma kullanmaktır. X_k , k . pozisyonunda k elemanını içeren permütasyon kümesi olsun. Örneğin $n = 3$ olduğu zaman kümeler aşağıdaki gibidir:

$$\begin{aligned} X_1 &= \{(1, 2, 3), (1, 3, 2)\} \\ X_2 &= \{(1, 2, 3), (3, 2, 1)\} \\ X_3 &= \{(1, 2, 3), (2, 1, 3)\} \end{aligned}$$

Bu kümeleri kullanarak toplam düzensizlik sayısı

$$n! - |X_1 \cup X_2 \cup \dots \cup X_n|,$$

olur yani bu birleşim kısımlarının büyüklüğünü hesaplamak yeterlidir. İçerme-dışarma sayesinde kesişim büyüklüklerini hesaplamak verimli bir şekilde yapılabilir. Örneğin, $n = 3$ ise $|X_1 \cup X_2 \cup X_3|$

$$\begin{aligned} & |X_1| + |X_2| + |X_3| - |X_1 \cap X_2| - |X_1 \cap X_3| - |X_2 \cap X_3| + |X_1 \cap X_2 \cap X_3| \\ &= 2 + 2 + 2 - 1 - 1 - 1 + 1 \\ &= 4, \end{aligned}$$

olur ve toplam çözüm sayısı $3! - 4 = 2$ idir.

Bu problem aynı zamanda içerme-dışarma olmadan da çözülebilir. $f(n)$, $\{1, 2, \dots, n\}$ kümesindeki düzensizlik miktarını versin. Aşağıdaki özyinelemeli formül kullanılabilir:

$$f(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ (n-1)(f(n-2) + f(n-1)) & n > 2 \end{cases}$$

Bu formül 1. elemanın düzensizlikte değişiklik yollarına göre bakarak bulunabiliriz. 1. elemanı değiştiren bir x elemanı seçmenin $n-1$ yolu vardır. Bu tip bir durumda iki seçenek vardır: The formula can be derived by considering the possibilities how the element 1 changes in the derangement. There are $n-1$ ways to choose an element x that replaces the element 1. In each such choice, there are two options:

1. *Seçenek:* 1. elemanı da x elemanının yerine koyarız. Bundan sonra amacımız $n-2$ elemandan bir düzensizlik oluşturmaktır.
2. *Seçenek:* x elemanının yerine 1. eleman hariç başka bir eleman koyarız. Şimdi $n-1$ tane elemandan oluşan bir düzensizlik oluşturmamız gerekir çünkü x elemanının 1. eleman ile değiştiremeyiz ve diğer elemanlar da yer değiştirmek zorundadır.

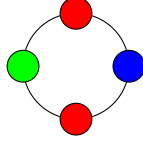
22.4 Burnside'in Lemma'sı

Burnside'in Lemma'sı simetrik kombinasyon gruplarını sadece bir defa sayabilmemizi sağlayan kombinasyon sayısını hesaplama yoludur. Burnside'in Lemma'sı bize kombinasyon sayısının kombinasyonun pozisyonunun n defa değiştirme yolu olduğu zaman n

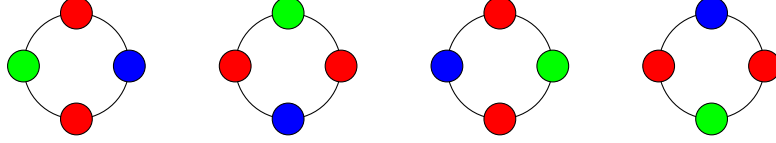
$$\sum_{k=1}^n \frac{c(k)}{n}$$

olduğunu ve k . yol uygulandığı zaman $c(k)$ tane kombinasyonun değişmediğini söyler.

Örneğin, n tane inciden oluşan kolye sayısını hesaplayalım. Her incinin olası m tane rengi vardır ve iki kolye belirli şekilde döndürülerek aynı oluyorsa bu ikisi simetriktir. Örneğin



kolyesinin aşağıdaki simetrik kolyeleri vardır:



Kolyenin pozisyonunu değiştirmenin n farklı yolu vardır çünkü saat yönüne doğru $0, 1, \dots, n-1$ adım döndürebiliriz. Eğer adım sayısı 0 ise bütün m^n kolye sayısı aynı kalır ve eğer adım sayısı 1 ise sadece her incisinin rengi aynı olan m kolye aynı kalır.

Genel olarak toplam adım sayısı k ise toplam

$$m^{\gcd(k,n)}$$

kolye aynı kalır. Burada $\gcd(k,n)$, k ve n elemanlarının en büyük ortak bölenidir. Bunun nedeni ise $\gcd(k,n)$ büyüklüğünde olan inci blokları birbirlerinin yerlerine geçecekler. Burnside'in Lemma'sına göre toplam kolye sayısı

$$\sum_{i=0}^{n-1} \frac{m^{\gcd(i,n)}}{n}.$$

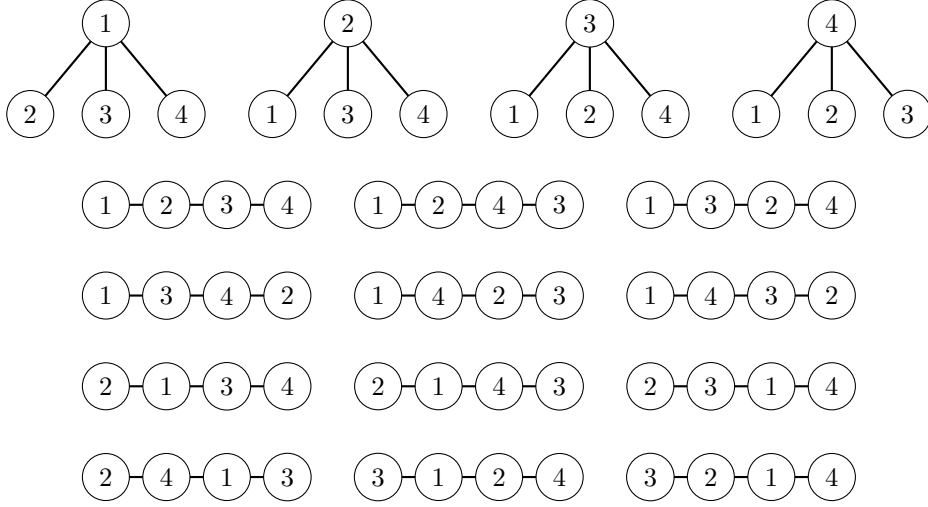
Örneğin 3 renkten oluşan 4 uzunluğundaki kolyelerin sayısı

$$\frac{3^4 + 3 + 3^2 + 3}{4} = 24.$$

22.5 Cayley'in Formülü

Cayley'in Formülü n tane düğümden oluşan n^{n-2} tane etiketli ağaç vardır. Düğümler $1, 2, \dots, n$ şeklinde etiketlenir ve iki ağacın yapısı ve etiketi farklı ise farklı olur.

Örneğin $n = 4$ ise toplam etiketli ağaç sayısı $4^{4-2} = 16$ olur:

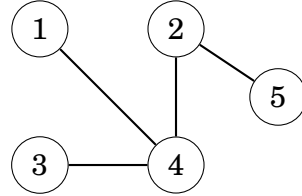


Şimdi Cayley'in Formülü'nün Prüfer kodlarından nasıl bulunabileceğine bakalım.

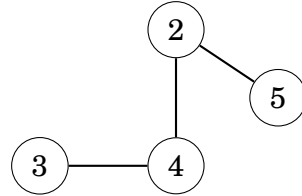
Prüfer kodu

Prüfer kodu etiketli bir ağacı gösteren $n-2$ tane sayıdan oluşan bir dizidir. Kod $n-2$ yaprağı ağaçtan silme işlemiyle oluşturulur. Her adımda en küçük etikete sahip yaprak kaldırılır ve tek komşusunun etiketi koda eklenir.

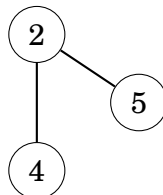
Örneğin, aşağıdaki çizgenin Prüfer kodunu bulalım.



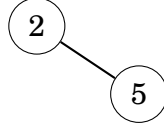
İlk başta 1. düğümü kaldırıp 4. düğümü koda ekleriz:



Sonra 3. düğümü kaldırıp 4. düğümü koda ekleriz:



En sonda 4. düğümü kaldırıp 2. düğümü koda ekleriz:



Böylece çizgenin Prüfer kodu $[4,4,2]$ olur.

Herhangi bir ağacın Prüfer kodu oluşturulabilir ve daha da önemlisi Prüfer koddan orijinal ağaç yeniden oluşturulabilir. Böylece n tane düğümden oluşan etiketli ağaç sayısı n^{n-2} olur ve bu da n büyüklüğündeki Prüfer kod sayısına eşittir.

Bölüm 23

Matris (Matrix)

Matris programlamada iki boyutlu diziye karşılık gelen bir matematik konseptidir. Örneğin

$$A = \begin{bmatrix} 6 & 13 & 7 & 4 \\ 7 & 0 & 8 & 2 \\ 9 & 5 & 4 & 18 \end{bmatrix}$$

3×4 büyüklüğündeki bir matristir yani 3 satır ve 4 sütundan oluşur. $[i, j]$ gösterimi matriste i . satırda ve j . sütunda bulunan elemanı gösterir. Örneğin $A[2, 3] = 8$ ve $A[3, 1] = 9$ olur.

Matrisin özel bir durumu $n \times 1$ büyüklüğünde tek boyutlu matris olan **vektör** olur. Örneğin

$$V = \begin{bmatrix} 4 \\ 7 \\ 5 \end{bmatrix}$$

3 eleman içeren bir vektördür.

A matrisinin A^T **transpozu** A 'nın sütunları ve satırları yer değiştirdiği zaman olur yani $A^T[i, j] = A[j, i]$:

$$A^T = \begin{bmatrix} 6 & 7 & 9 \\ 13 & 0 & 5 \\ 7 & 8 & 4 \\ 4 & 2 & 18 \end{bmatrix}$$

Eğer satır ve sütun sayısı eşit ise bu matrix bir **kare matrisi** olur. Örneğin aşağıdaki matris bir kare matrisidir:

$$S = \begin{bmatrix} 3 & 12 & 4 \\ 5 & 9 & 15 \\ 0 & 2 & 4 \end{bmatrix}$$

23.1 Operasyonlar

A ve B matrislerinin toplamı $A + B$ eğer matrisler aynı boyutlarda ise geçerlidir. Sonuç olarak oluşan matristeki her eleman, o elemanın pozisyonun A ve B 'deki elemanların toplamıdır.

Örneğin

$$\begin{bmatrix} 6 & 1 & 4 \\ 3 & 9 & 2 \end{bmatrix} + \begin{bmatrix} 4 & 9 & 3 \\ 8 & 1 & 3 \end{bmatrix} = \begin{bmatrix} 6+4 & 1+9 & 4+3 \\ 3+8 & 9+1 & 2+3 \end{bmatrix} = \begin{bmatrix} 10 & 10 & 7 \\ 11 & 10 & 5 \end{bmatrix}.$$

Bir A matrisini x değeriyle çarpmak A matrisinin her elemanını x ile çarpmaktır. Örneğin

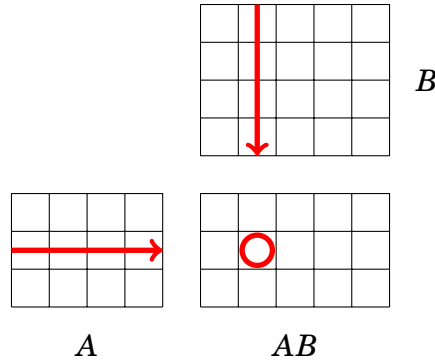
$$2 \cdot \begin{bmatrix} 6 & 1 & 4 \\ 3 & 9 & 2 \end{bmatrix} = \begin{bmatrix} 2 \cdot 6 & 2 \cdot 1 & 2 \cdot 4 \\ 2 \cdot 3 & 2 \cdot 9 & 2 \cdot 2 \end{bmatrix} = \begin{bmatrix} 12 & 2 & 8 \\ 6 & 18 & 4 \end{bmatrix}.$$

Matris Çarpımı

AB çarpımı, eğer A , $a \times n$ büyüklüğünde ve B $n \times b$ büyüklüğünde ise yeni A 'nin genişliği B 'nin yüksekliğine eşitse ve tam tersi de eşitse tanımlıdır. Sonuç olarak $a \times b$ büyüklüğünde bir matris oluşur ve bu matrisin elemanları

$$AB[i,j] = \sum_{k=1}^n A[i,k] \cdot B[k,j].$$

formülü ile hesaplanır. Buradaki fikir AB 'deki her elemanın A ve B elemanların aşağıdaki resme göre çarpımlarının toplamına eşit olur:



Örneğin

$$\begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} \cdot \begin{bmatrix} 1 & 6 \\ 2 & 9 \end{bmatrix} = \begin{bmatrix} 1 \cdot 1 + 4 \cdot 2 & 1 \cdot 6 + 4 \cdot 9 \\ 3 \cdot 1 + 9 \cdot 2 & 3 \cdot 6 + 9 \cdot 9 \\ 8 \cdot 1 + 6 \cdot 2 & 8 \cdot 6 + 6 \cdot 9 \end{bmatrix} = \begin{bmatrix} 9 & 42 \\ 21 & 99 \\ 20 & 102 \end{bmatrix}.$$

Matris çarpımı birleşme özelliğine sahiptir yani $A(BC) = (AB)C$ olur fakat değişme özelliğine sahip değildir yani $AB \neq BA$ olmak zorunda değildir.

Birim matris (identity matrix) köşegenlerdeki her elemanın 1 olup geri kalan elemanların 0 olduğu bir kare matrisidir. Örneğin aşağıdaki matris 3×3 boyutunda bir birim matristir:

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Matrisi bir birim matrisle çarpmak sonucu değiştirmemektedir. Örneğin

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} x! \text{ and } x! \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix}.$$

Düz bir algoritma kullanarak $n \times n$ büyüklüğündeki iki matrisin matris çarpımını $O(n^3)$ zamanda hesaplayabiliriz. Bununla beraber matris çarpımı için daha verimli algoritmalar vardır¹ fakat bunlar genel olarak teorik ilgi içindir ve bu tip algoritmalar rekabetçi programlamada ve olimpiyatta gerekli değildir.

Matris Kuvveti

A matrisinin A^k kuvveti eğer A bir kare matrisi ise geçerlidir. Tanımı matris çarpımından gelir:

$$A^k = \underbrace{A \cdot A \cdot A \cdots A}_{k \text{ times}}$$

Örneğin

$$\begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^3 = \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix} \cdot \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix} \cdot \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix} = \begin{bmatrix} 48 & 165 \\ 33 & 114 \end{bmatrix}.$$

Bunla beraber A^0 bir birim matrisidir. Örneğin

$$\begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

A^k matrisi verimli bir şekilde $O(n^3 \log k)$ zamanda Bölüm 21.2'deki algoritma kullanılarak bulunabilir. Örneğin

$$\begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^8 = \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^4 \cdot \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^4.$$

Determinant

A matrisinin **determinantı** $\det(A)$ eğer A bir kare matrisi ise tanımlıdır. Eğer A 1×1 büyüklüğünde ise $\det(A) = A[1,1]$ olur. Daha büyük bir matrisin determinantı özyinelemeli bir şekilde

$$\det(A) = \sum_{j=1}^n A[1,j]C[1,j],$$

, formülü ile hesaplanabilir. Burada $C[i,j]$, A 'nın $[i,j]$ 'deki elemanının **kofaktörüdür**. Kofaktör

$$C[i,j] = (-1)^{i+j} \det(M[i,j])$$

¹Bu tip ilk algoritma, 1969 yılında basılan Strassen'in algoritmasıdır [63]. Algoritmanın zaman karmaşıklığı $O(n^{2.80735})$ idi. Şu anki en iyi algoritma [27] olup $O(n^{2.37286})$ zamanda çalışır.

kullanarak bulunabilir. Burada $M[i, j]$, A 'dan i . satır ve j . sütun çıkartarak bulunabilir. Kofaktördeki $(-1)^{i+j}$ katsayısından dolayı geriye kalan her determinant pozitif ve negatiftir. Örneğin

$$\det\begin{pmatrix} 3 & 4 \\ 1 & 6 \end{pmatrix} = 3 \cdot 6 - 4 \cdot 1 = 14$$

and

$$\det\begin{pmatrix} 2 & 4 & 3 \\ 5 & 1 & 6 \\ 7 & 2 & 4 \end{pmatrix} = 2 \cdot \det\begin{pmatrix} 1 & 6 \\ 2 & 4 \end{pmatrix} - 4 \cdot \det\begin{pmatrix} 5 & 6 \\ 7 & 4 \end{pmatrix} + 3 \cdot \det\begin{pmatrix} 5 & 1 \\ 7 & 2 \end{pmatrix} = 81.$$

A 'nın determinanı bize I 'nın birim matris olduğu $A \cdot A^{-1} = I$ sonucunu verebilecek bir A^{-1} ters matrisinin olup olmadığını ve söyler. Eğer $\det(A) \neq 0$ olursa A^{-1} vardır ve

$$A^{-1}[i, j] = \frac{C[j, i]}{\det(A)}.$$

formülü ile hesaplanır. Örneğin

$$\underbrace{\begin{pmatrix} 2 & 4 & 3 \\ 5 & 1 & 6 \\ 7 & 2 & 4 \end{pmatrix}}_A \cdot \underbrace{\frac{1}{81} \begin{pmatrix} -8 & -10 & 21 \\ 22 & -13 & 3 \\ 3 & 24 & -18 \end{pmatrix}}_{A^{-1}} = \underbrace{\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}}_I.$$

23.2 Linear Tekrar (Linear Recurrences)

Linear tekrar $f(n)$ fonksiyonu olup ilk değerleri $f(0), f(1), \dots, f(k-1)$ olur ve daha büyük değerler özyinelemeli bir şekilde

$$f(n) = c_1 f(n-1) + c_2 f(n-2) + \dots + c_k f(n-k)$$

formülü ile hesaplanabilir. c_1, c_2, \dots, c_k burada sabit katsayılardır.

Dinamik programlama kullanarak herhangi bir $f(n)$ değeri $O(kn)$ zamanda hesaplanabilir. Bunu da bütün $f(0), f(1), \dots, f(n)$ değerlerini art arda hesaplayarak yapabiliriz. Fakat k küçükse $f(n)$ değerini daha verimli hesaplamayı matris operasyonları ile $O(k^3 \log n)$ zamanda hesaplayabiliriz.

Fibonacci Sayıları

Linear tekrar örneklerinden biri Fibonacci sayılarını tanımlayan aşağıdaki fonksiyondur:

$$\begin{aligned} f(0) &= 0 \\ f(1) &= 1 \\ f(n) &= f(n-1) + f(n-2) \end{aligned}$$

Bu durumda $k = 2$ ve $c_1 = c_2 = 1$ olur.

Fibonacci sayılarını verimli hesaplamamanın bir yolu ise Fibonacci formülünü 2×2 büyüklüğünde bir X kare matrisiyle göstererek yapabiliriz:

$$X \cdot \begin{bmatrix} f(i) \\ f(i+1) \end{bmatrix} = \begin{bmatrix} f(i+1) \\ f(i+2) \end{bmatrix}$$

Böylece $f(i)$ ve $f(i+1)$ değerleri bize "girdide" X için verilir ve X , $f(i+1)$ ve $f(i+2)$ değerlerini bunlardan hesaplar. Bu tip bir matris

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}.$$

Örneğin

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} f(5) \\ f(6) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 5 \\ 8 \end{bmatrix} = \begin{bmatrix} 8 \\ 13 \end{bmatrix} = \begin{bmatrix} f(6) \\ f(7) \end{bmatrix}.$$

Böylece $f(n)$ değerini

$$\begin{bmatrix} f(n) \\ f(n+1) \end{bmatrix} = X^n \cdot \begin{bmatrix} f(0) \\ f(1) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

formülü ile hesaplayabiliriz. X^n değeri $O(\log n)$ zamanda hesaplanabilir ve böylece $f(n)$ değeri de $O(\log n)$ hesaplanabilir.

Genel Durum

$f(n)$ 'in herhangi bir linear tekrar olduğu genel bir duruma bakalım. Yine buradaki amacımız bir X matrisi oluşturmaktır. Burada X matrisi

$$X \cdot \begin{bmatrix} f(i) \\ f(i+1) \\ \vdots \\ f(i+k-1) \end{bmatrix} = \begin{bmatrix} f(i+1) \\ f(i+2) \\ \vdots \\ f(i+k) \end{bmatrix}.$$

Bu tip matris

$$X = \begin{bmatrix} 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 \\ c_k & c_{k-1} & c_{k-2} & c_{k-3} & \cdots & c_1 \end{bmatrix}$$

şeklinde olmalıdır. İlk $k-1$ satırda, bir tane eleman hariç hepsi 0'dır ve o geriye kalan eleman ise 1'dir. Bu satırlar $f(i)$ 'i $f(i+1)$ ile $f(i+1)$ 'i $f(i+2)$ ile ve geri kalanları benzer şekilde değiştirir. Son satır yeni $f(i+k)$ değerini hesaplamak için tekrarın katsayılarını bulundurmaz.

Şimdi $f(n)$, $O(k^3 \log n)$ zamanda

$$\begin{bmatrix} f(n) \\ f(n+1) \\ \vdots \\ f(n+k-1) \end{bmatrix} = X^n \cdot \begin{bmatrix} f(0) \\ f(1) \\ \vdots \\ f(k-1) \end{bmatrix}.$$

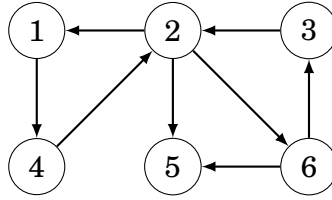
formülü kullanılarak hesaplanabilir.

23.3 Çizgeler ve Matrisler

Yolları Saymak

Bir çizgenin komşuluk matrisinin üslerinin ilginç bir özelliği vardır. V ağırlıksız bir çizgenin komşuluk matrisi ise V^n matrisi de çizgedeki düğümler arası n kenara sahip yolların sayısını tutar.

Örneğin



çizgesinde komşuluk matrisi

$$V = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}.$$

gibi olur. Şimdi örneğin

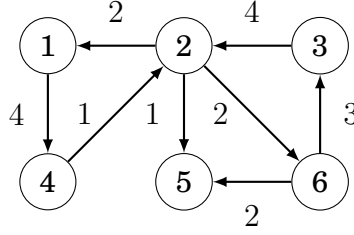
$$V^4 = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 0 \\ 2 & 0 & 0 & 0 & 2 & 2 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

düğümler arasında 4 kenar uzunluğunda olan yol sayısını tutar. Örneğin $V^4[2,5] = 2$ olur çünkü 2. düğümden 5. düğüme 4 kenarlı iki yol vardır: $2 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 5$ ve $2 \rightarrow 6 \rightarrow 3 \rightarrow 2 \rightarrow 5$.

En Kısa Yollar

Benzer bir fikri ağırlıklı bir çizgede kullanarak her düğüm çifti için n kenardan oluşan en kısa yolun uzunluğunu bulabiliriz. Bunu yapmak için için matris çarpımını farklı bir yolda kullanırız ve böylece yol sayısını hesaplamak yerine yolların uzunluğunu minimize etmeye çalışırız.

Örneğin aşağıdaki çizgede:



Bir komşuluk matrisi oluşturalım. ∞ 'nin bulunması orada kenarın bulunmadığını gösterir ve geriye kalan matrisin değerleri de kenarların ağırlıkları olur. Matris

$$V = \begin{bmatrix} \infty & \infty & \infty & 4 & \infty & \infty \\ 2 & \infty & \infty & \infty & 1 & 2 \\ \infty & 4 & \infty & \infty & \infty & \infty \\ \infty & 1 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 3 & \infty & 2 & \infty \end{bmatrix}.$$

Aşağıdaki formül yerine

$$AB[i, j] = \sum_{k=1}^n A[i, k] \cdot B[k, j]$$

Şimdi matris çarpımı için

$$AB[i, j] = \min_{k=1}^n A[i, k] + B[k, j]$$

bu formülü kullanabiliriz ki böylece toplam yerine minimum hesaplayıp çarpım yerine elemanların toplamını buluruz. Bu değişiklikten sonra matris kuvvetleri çizgedeki en kısa yollara karşılık gelir.

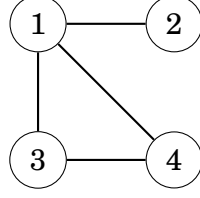
Örneğin

$$V^4 = \begin{bmatrix} \infty & \infty & 10 & 11 & 9 & \infty \\ 9 & \infty & \infty & \infty & 8 & 9 \\ \infty & 11 & \infty & \infty & \infty & \infty \\ \infty & 8 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 12 & 13 & 11 & \infty \end{bmatrix},$$

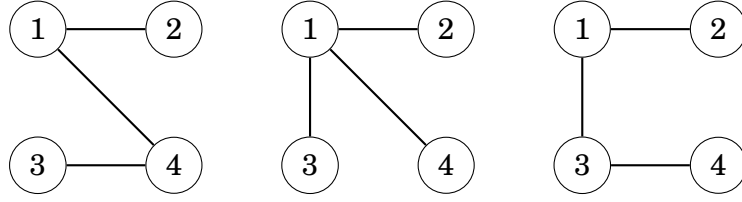
2. düğümden 5. düğüme 4 kenar olan bir yolun en kısa uzunluğunun 8 olduğunu buluruz. Bu yol $2 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 5$ şeklinde olur.

Kirchhoff's theorem

Kirchhoff'un Teoremi bir çizgenin toplam kapsayan ağaç sayısını özel bir matrisin determinantı şeklinde hesaplamamızı sağlar. Örneğin



çizgesinin 3 tane dolaşımli ağacı vardır:



Dolaşımli ağaç sayısını hesaplamak için bir **L laplas matrisi** oluştururuz ve burada $L[i, i]$ i düğümünün derecesi olur. Eğer i ve j düğümleri arasında kenar varsa $L[i, j] = -1$ ve eğer yoksa $L[i, j] = 0$ olur. Yukarıdaki çizgenin laplas matrisi aşağıdaki gibi olur:

$$L = \begin{bmatrix} 3 & -1 & -1 & -1 \\ -1 & 1 & 0 & 0 \\ -1 & 0 & 2 & -1 \\ -1 & 0 & -1 & 2 \end{bmatrix}$$

Toplam dolaşımli ağaç sayısının L 'den herhangi bir sütunu ve satırı çıkardığımızda olan matrisin determinantına eşittir. Örneğin, eğer ilk satırı ve sütunu çıkardığımızda sonuç

$$\det \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & -1 \\ 0 & -1 & 2 \end{pmatrix} = 3$$

olur. L 'den hangi sütunu ve satırı çıkarırsak çıkaralım determinant aynıdır.

Bölüm 22.5'teki Cayley'in Formülü aslında Kirchhoff'un Teoremi'nin özel bir durumudur çünkü n düğümden oluşan tam bir çizgede

$$\det \begin{pmatrix} n-1 & -1 & \cdots & -1 \\ -1 & n-1 & \cdots & -1 \\ \vdots & \vdots & \ddots & \vdots \\ -1 & -1 & \cdots & n-1 \end{pmatrix} = n^{n-2}.$$

Bölüm 24

Olasılık (Probability)

Olasılık 0'dan 1'e kadar olan bir gerçek sayı olup bir olayın gerçekleşme ihtimalini verir. Eğer olay kesinlikle gerçekleşecekse olasılığı 1 ve eğer imkansızsa olasılığı 0'dır. Olayın olasılığı $P(\dots)$ şeklinde gösterilir ve üç nokta olayı gösterir.

Örneğin zar atınca gelecek sayı 1'den 6'ya kadar bir tam sayıdır ve her sayının olasılığı $1/6$ 'dır. Örneğin aşağıdaki olasılıkları hesaplayabiliriz:

- $P(\text{"the outcome is 4"}) = 1/6$
- $P(\text{"the outcome is not 6"}) = 5/6$
- $P(\text{"the outcome is even"}) = 1/2$

24.1 Hesaplama

Bir olayın olasılığını hesaplamak için ya kombinatorik kullanırız ya da olayı sağlayacak işlemi simüle ederiz. Örneğin karıştırılmış bir deste iskambil kağıdı arasından aynı değere sahip 3 kart seçme olasılığını bulalım (örneğin ♠8, ♣8 ve ♦8).

1. Method

Olasılığı

$$\frac{\text{istenen durum sayısı}}{\text{toplam olası durum sayısı}}$$

formülü ile bulabiliriz. Bu problemde istenen durumlar seçtiğimiz her karttaki değerlerin aynı olmasıdır. Bunu sağlayan $13 \binom{4}{3}$ durum vardır çünkü 13 tane kart değeri vardır bunları seçmenin her değer için $\binom{4}{3}$ yolu vardır çünkü aynı değere sahip 4 karttan 3 tanesini seçeriz.

Toplam $\binom{52}{3}$ durum vardır çünkü 52 kart arasından 3 kart seçeriz. Böylece bu olayın gerçekleşme olasılığı

$$\frac{13 \binom{4}{3}}{\binom{52}{3}} = \frac{1}{425}.$$

2. Method

Olasılığı hesaplamamızın başka bir yolu ise olayı sağlayan işlemi simüle etmektir. Örneğin, üç kart çekeriz yani işlemimiz üç adımdan oluşur. Her adımın istediğimiz gibi olması lazım.

İlk kartı seçmek kesinlikle başarılıdır çünkü herhangi bir sınırlamamız yoktur. İkinci adımda ise olasılığımız $3/51$ olur çünkü kalan 51 kart arasından sadece 3 tanesi birinci kartla aynı değere sahiptir. Benzer şekilde 3. kart için de $2/50$ ihtimalimiz vardır.

Bütün işlemin istediğimiz gibi olma olasılığı:

$$1 \cdot \frac{3}{51} \cdot \frac{2}{50} = \frac{1}{425}.$$

24.2 Olaylar

Olasılık teorisi içindeki bir olay bir küme şeklinde gösterilebilir:

$$A \subset X,$$

burada X bütün olası durumları içerir ve A ise bu olası durumlardan oluşan bir alt kümedir. Örneğin zar atarken olası durumlar

$$X = \{1, 2, 3, 4, 5, 6\}.$$

Şimdi örneğin, "görünen yüzün çift sayı olma" olayı ise aşağıdaki kümeye karşılık gelir:

$$A = \{2, 4, 6\}.$$

Her x olası durumu için bir $p(x)$ ihtimali atanır. Sonrasında bir A etkinliğinin olasılığı $P(A)$ içinde bulundurduğu olasılıkların toplamı olarak aşağıdaki formülle hesaplanabilir

$$P(A) = \sum_{x \in A} p(x).$$

Örneğin zar attığımızda her x olası durumu için $p(x) = 1/6$ olur yani "görünen yüzün çift gelme" olayının ihtimali ise

$$p(2) + p(4) + p(6) = 1/2.$$

X 'teki olası durumların toplam ihtimalleri 1 olmalıdır yani $P(X) = 1$.

Olasılık teorisindeki olaylar küme olduğu için onları standart küme operasyonları ile değiştirebiliriz.

- **Tümleyen** \bar{A} "A gerçekleşmeyecek" anlamına gelir. Örneğin $A = \{2, 4, 6\}$ 'nın tümleyeni $\bar{A} = \{1, 3, 5\}$ olur.
- **Birleşim** $A \cup B$ ise "A veya B gerçekleşecek" anlamına gelir. Örneğin $A = \{2, 5\}$ ve $B = \{4, 5, 6\}$ kümelerinin birleşimi $A \cup B = \{2, 4, 5, 6\}$ olur.
- **Kesişim** $A \cap B$ ise "A ve B gerçekleşecek" anlamına gelir. Örneğin $A = \{2, 5\}$ ve $B = \{4, 5, 6\}$ kümelerinin kesişimi $A \cap B = \{5\}$ olur.

Tümleyen

Tümleyen \bar{A} olma olasılığı aşağıdaki formül ile hesaplanır:

$$P(\bar{A}) = 1 - P(A).$$

Bazen bir problemi kolay bir şekilde tümleyenini kullanarak zıt problemini çözerek çözebiliriz. Örneğin 10 defa zar attığımızda görünen yüze en az bir defa 6 gelme olasılığı

$$1 - (5/6)^{10}.$$

Burada $5/6$ olasılığı gelen sayının 6 olmayan bir atış olur ve $(5/6)^{10}$ ise gelen sayının hiçbir zaman 6 olmadığı 10 atış olur. Bu bulduğumuz çözümün tümleyeni orijinal problemin cevabıdır.

Birleşim

$A \cup B$ birleşiminin olasılığı

$$P(A \cup B) = P(A) + P(B) - P(A \cap B)$$

formülü ile hesaplanır. Örneğin zar atıldığı zaman

$$A = \text{"sonuç çift sayıdır"}$$

ve

$$B = \text{"sonuç 4'ten küçüktür"}$$

olaylarının birleşimi

$$A \cup B = \text{"sonuç çift veya 4'ten küçüktür"}$$

olur ve olasılığı ise

$$P(A \cup B) = P(A) + P(B) - P(A \cap B) = 1/2 + 1/2 - 1/6 = 5/6.$$

Eğer A ve B olayları **ayrık** kümelerse yani $A \cap B$ boşsa, $A \cup B$ olayının gerçekleşme olasılığı

$$P(A \cup B) = P(A) + P(B)$$

olur.

Koşullu Olasılık

Koşullu olasılık

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

B 'nin gerçekleşeceği kabul edilerek A 'nın gerçekleşme olasılığıdır. Bu yüzden A 'nın gerçekleşme olasılığı hesaplanırken sadece B 'ye ait çıktılar hesaplanır.

Önceki kümelerde

$$P(A|B) = 1/3$$

çünkü B 'nin çıktıları $\{1, 2, 3\}$ ve bunlardan sadece birisi çifttir. Bu çift sayı çıkma olasılığıdır çünkü çıktının $1 \dots 3$ arasında olacağını biliyoruz.

Kesişim

Koşullu olasılık kullanılarak kesişim $A \cap B$ olasılığını

$$P(A \cap B) = P(A)P(B|A)$$

formülü ile bulabiliriz. A ve B olayları eğer

$$P(A|B) = P(A) \text{ ve } P(B|A) = P(B)$$

ise yani B 'nin gerçekleşme olasılığı A 'yı etkilemiyorsa veya tam tersinde de etkilemiyorsa bu olaylar **bağımsızdır**. Bu durumda kesişimin olasılığı

$$P(A \cap B) = P(A)P(B).$$

Örneğin desteden kart çekerken

$$A = \text{"bir sinek kağıdı"}$$

ve

$$B = \text{"değeri 4"}$$

olması olayları birbirinden bağımsızdır. Bu yüzden

$$A \cap B = \text{"kart sinek 4'lüsü"}$$

olayının gerçekleşme olasılığı

$$P(A \cap B) = P(A)P(B) = 1/4 \cdot 1/13 = 1/52.$$

24.3 Rastgele Değişkenler

Rastgele değişken rastgele bir işlem sırasında oluşturulmuş bir değerdir. Örneğin iki zar attığımızda olası rastgele bir değişken

$$X = \text{"gelen sayıların toplamı"}$$

olur. Örneğin gelen sayıları $[4, 6]$ (ilk attığımızda 4 ve ikinci attığımızda 6 geldi) ise X değeri 10 olur.

$P(X = x)$ ise bir rastgele X değişkeninin değerinin x olma olasılığını gösterir. Örneğin iki zar attığımızda $P(X = 10) = 3/36$ olur çünkü toplam olası durum sayısı 36'dır ve 10 toplamını elde etmek için 3 olası durum vardır: $[4, 6]$, $[5, 5]$ ve $[6, 4]$.

Beklenen Değer (Expected value)

Beklenen değer $E[X]$ ise rastgele X değişkeninin ortalama değerini verir. Beklenen değer

$$\sum_x P(X = x)x$$

toplamı ile hesaplanır ki burada x bütün olası X değerlerinden geçer.

Örneğin zar attığımızda beklenen değer

$$1/6 \cdot 1 + 1/6 \cdot 2 + 1/6 \cdot 3 + 1/6 \cdot 4 + 1/6 \cdot 5 + 1/6 \cdot 6 = 7/2.$$

Beklenen değerlerin işe yarar bir özelliği **linearliği** idir. Yani $E[X_1 + X_2 + \dots + X_n]$ toplamı her zaman $E[X_1] + E[X_2] + \dots + E[X_n]$ toplamına eşittir. Bu formül, rastgele değişkenler birbirine bağlı olsa bile geçerlidir.

Örneğin iki zar attığımızda beklenen değer

$$E[X_1 + X_2] = E[X_1] + E[X_2] = 7/2 + 7/2 = 7.$$

Şimdi n tane topun rastgele n kutuya yerleştirildiğini düşünelim. Amacımız boş kutu sayısının beklenen değerini bulmak olsun. Her top, herhangi bir kutuya eşit olasılıkla yerleştirilebilir. Örneğin $n = 2$ ise olasılıklar aşağıdaki gibi olur:



Bu durumda boş kutu sayısının beklenen değeri

$$\frac{0 + 0 + 1 + 1}{4} = \frac{1}{2}.$$

Genel durumda tek bir boş kutunun olma ihtimali

$$\left(\frac{n-1}{n}\right)^n$$

çünkü herhangi bir topun oraya koyulmaması gerekir. Böylece linearlik kullanarak boş kutu sayısının beklenen değeri

$$n \cdot \left(\frac{n-1}{n}\right)^n.$$

Dağılımlar (Distributions)

Rastgele bir değişken olan x 'in **dağılımı** X 'in sahip olabileceği her değer ihtimalini gösterir. Dağılım $P(X = x)$ değerlerinden oluşur. Örneğin iki zar atıldığı zaman toplamalarının dağılımı:

x	2	3	4	5	6	7	8	9	10	11	12
$P(X = x)$	1/36	2/36	3/36	4/36	5/36	6/36	5/36	4/36	3/36	2/36	1/36

Sürekli dağılımda, rastgele x değişkenin n tane olası değeri olup bu değerler $a, a+1, \dots, b$ olur ve her değer ihtimali ise $1/n$ idir. Örneğin bir zar atıldığı zaman her x değeri için $a = 1$, $b = 6$ ve $P(X = x) = 1/6$.

X 'in sürekli dağılımdaki beklenen değeri

$$E[X] = \frac{a+b}{2}.$$

Binom dağılımında, n tane deneme yapılır ve bir denemenin başarılı olma ihtimali p olur. Rastgele X değişkeni toplam başarılı deneme sayısını sayar. Bir x değerinin ihtimali ise

$$P(X = x) = p^x(1 - p)^{n-x} \binom{n}{x}$$

ve burada p^x ve $(1 - p)^{n-x}$ başarılı ve başarısız denemelere karşılık gelir. $\binom{n}{x}$ ise denemelerin sırasını seçme yollarını gösterir.

Örneğin bir zarı 10 defa attığımız zaman tam 3 defa 6 sayısının gelme ihtimali $(1/6)^3(5/6)^7 \binom{10}{3}$ olur.

Binom dağılımında X 'in beklenen değeri

$$E[X] = pn.$$

Geometrik dağılımında, bir denemenin başarılı olma ihtimali p olur ve ilk defa başarılı olana kadar devam ederiz. X rastgele değişkeni ise gereken deneme sayısını sayar ve bir x değerinin olasılığı

$$P(X = x) = (1 - p)^{x-1}p$$

olur ve burada $(1 - p)^{x-1}$ ise başarısız deneme sayısına karşılık gelir ve p ise de ilk başarılı denemeye karşılık gelir.

Örneğin 6 sayısı gelene kadar zar atarsak toplam 4 defa zar attığımız olasılık $(5/6)^3 1/6$.

X 'in geometrik dağılımdaki beklenen değeri

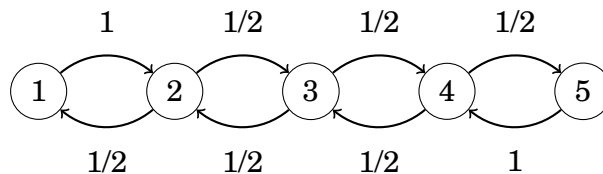
$$E[X] = \frac{1}{p}.$$

24.4 Markov Zincirleri

Markov zinciri durumlardan ve geçişlerden oluşan bir rastgele işlemdir. Her durumda diğer duruma geçme olasılıklarını biliyoruz. Markov zinciri bir çizge olarak gösterilebilir düğümler durum ve kenarlar geçiş olarak gösterilir.

Örnek olarak, n katlı bir binada 1. katta olduğumuzu düşünelim. Her adımda her seferinde ya bir kat yukarı ya da bir kat aşağı yürürüz. Burada 1. kattan her zaman bir adım yukarı ve n . kattan bir kat aşağı inmemiz gerekir. k adım sonra m . katta olma ihtimalimiz nedir?

Bu problemde binanın her kat, Markov zincirinin bir durumu olarak kabul edilir. Örneğin $n = 5$ ise çizge aşağıdaki gibidir:



Bir Markov zincirinin olasılık ihtimali bir $[p_1, p_2, \dots, p_n]$ zinciridir ve burada p_k , şu anki durumun k olduğu ihtimali verir. $p_1 + p_2 + \dots + p_n = 1$ formülü her zaman doğrudur.

Yukarıdaki senaryoda, ilk dağılım $[1, 0, 0, 0, 0]$ olur çünkü 1. katta başlarız. Sonraki dağılım $[0, 1, 0, 0, 0]$ olur çünkü 1. kattan sadece 2. kata gidebiliriz. Bundan sonra ya bir kat yukarı ya da bir kat aşağı gidebiliriz ve böylece sonraki dağılım $[1/2, 0, 1/2, 0, 0]$ olur ve böylece devam eder.

Markov zincirini verimli bir şekilde simüle etmenin yolu dinamik programlama kullanmaktır. Buradaki fikir olasılık dağılımını tutup her adımda hareket edebileceğimiz her ihtimalden geçeriz. Bu methodu kullanarak m adımlık bir yürüyüşü $O(n^2 m)$ zamanda yapabiliriz.

Markov zincirinin geçişleri aynı zamanda olasılık dağılımını güncelleyen bir matris şeklinde de gösterebilir. Yukarıdaki senaryoda matris

$$\begin{bmatrix} 0 & 1/2 & 0 & 0 & 0 \\ 1 & 0 & 1/2 & 0 & 0 \\ 0 & 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 1/2 & 0 & 1 \\ 0 & 0 & 0 & 1/2 & 0 \end{bmatrix}.$$

Bir olasılık dağılımını bu matrisle çarptığımızda her aım yeni dağılımı alırız. Örneğin $[1, 0, 0, 0, 0]$ dağılımından $[0, 1, 0, 0, 0]$ dağılıma gideriz.

$$\begin{bmatrix} 0 & 1/2 & 0 & 0 & 0 \\ 1 & 0 & 1/2 & 0 & 0 \\ 0 & 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 1/2 & 0 & 1 \\ 0 & 0 & 0 & 1/2 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

Matris kuvvetlerini verimli bir şekilde hesaplayarak m adımdan sonra dağılımı $O(n^3 \log m)$ zamanda hesaplayabiliriz.

24.5 Rastgele Algoritmalar

Bazen bir problemi çözerken problem olasılıklarla alakalı olmasa bile rastgeleliği kullanabiliriz. Bir **rastgele algoritma** rastgelelik üzerine kurulu bir algoritmadır.

Monte Carlo algoritması bazen yanlış cevap verebilen bir rastgele algoritmadır. Bu tip algoritmaların işe yarar olması için yanlış cevap verme olasılığının düşük olması gerekir.

Las Vegas algoritması her zaman doğru cevap verip çalışma süresi rastgele olan bir rastgele algoritmadır. Buradaki amaç yüksek ihtimale sahip verimli bir algoritma yapmaktır.

Şimdi rastgelelik kullanarak 3 örnek problem çözeceğiz.

Sıra İstatistikleri

Bir dizinin k . sıra istatistiği diziyi artan sırada sıraladıktan sonra k . pozisyonda bulunan elemana karşılık gelir. Herhangi bir sıra istatistiğini $O(n \log n)$ sürede diziyi sıralayarak hesaplamak kolaydır fakat sadece bir eleman bulmak için bütün diziyi sıralamak gerekli midir?

Bir rastgele algoritma kullanarak diziyi sıralamadan sıra istatistiğini bulabiliriz. **Hızlı seçme** (quickselect) algoritması¹ bir Las Vegas algoritmasıdır: Çalışma zamanı genel olarak $O(n)$ olur ve en kötü durumda $O(n^2)$ olur.

Algoritma diziden rastgele bir x eleman seçer ve x 'ten küçük elemanları dizinin sol tarafına ve diğer kalan elemanlar dizinin sağ kısmına yerleştirir. n elemanlı bir dizide $O(n)$ zamanda çalışır. Diyelim ki sol tarafta a eleman ve sağ tarafta b eleman bulunduğunu varsayalım. Eğer $a = k$ olursa x elemanı k . sıra istatistiği olur. Aksi durumda $a > k$ ise özyinelemeli bir şekilde sol kısım için k . sıra istatistiği buluruz. Eğer $a < k$ ise özyinelemeli bir şekilde sağ parçada $r = k - a$ olan r . sıra istatistiğini buluruz. Arama eleman bulunana kadar benzer bir şekilde ilerler.

Her x elemanı rastgele seçildiği zaman, dizinin büyüklüğü her adımda yarıya iner ve böylece k . sıra istatistiğinin zaman karmaşıklığı

$$n + n/2 + n/4 + n/8 + \dots < 2n = O(n).$$

Algoritma en kötü durumda yine $O(n^2)$ zaman gerektirir çünkü seçilen x elemanın dizideki en büyük veya en küçük olabilir ve bu durumda $O(n)$ adım gerekir. Fakat bu ihtimal o kadar küçüktür pratikte neredeyse hiç gerçekleşmez.

Matris Çarpımını Doğrulama

Sonraki problememiz A , B ve C matrislerinin büyüklüğünün $n \times n$ olduğu zaman $AB = C$ sağlayıp sağlamadığını *doğrulamaktır*. Tabii ki bu soruyu AB 'nin çarpımını bularak (ki bunu basit algoritmalar kullanarak $O(n^3)$ zamanda yaparız.) yapabiliriz ama tabii bunu en baştan hesaplamak yerine çözümü doğrulamak daha kolay olacaktır.

Bu soruyu Monte Carlo algoritması² ile çözebiliriz ve bu algoritmanın zaman karmaşıklığı sadece $O(n^2)$ idir. Fikir basittir: n elemandan oluşan rastgele bir X vektörü oluştururuz ve ABX ve CX matrislerini hesaplarız. Eğer $ABX = CX$ ise $AB = C$ olduğunu söyleriz ve eğer değilse $AB \neq C$ deriz.

Algoritmanın zaman karmaşıklığı $O(n^2)$ olur çünkü ABX ve CX matrislerini $O(n^2)$ zamanda hesaplayabiliriz. ABX matrisini verimli bir şekilde $A(BX)$ gösterimi ile hesaplayabiliriz ki böylece sadece iki tane $n \times n$ ve $n \times 1$ büyüklüğündeki matrislerin çarpımı gerekir.

Algoritmanın dezavantajı ise $AB = C$ olduğunu söylerken küçük bir ihtimalle

¹1961'de C. A. R. Hoare ortalamada verimli olan iki algoritma yayınladı: **hızlı sıralama** [36] diziyi sıralamak için ve **hızlı seçme** [37] sıra istatistiğini bulmak için.

²R. M. Freivalds bu algoritmayı 1977'de yayınlamıştır [26] ve bu bazen **Freivalds'ın Algoritması** adında geçer.

hata yapabiliyor olmasıdır. Örneğin

$$\begin{bmatrix} 6 & 8 \\ 1 & 3 \end{bmatrix} \neq \begin{bmatrix} 8 & 7 \\ 3 & 2 \end{bmatrix},$$

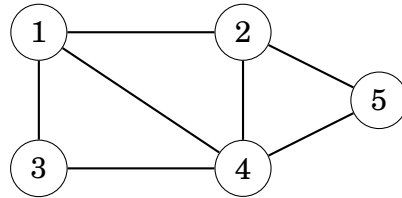
but

$$\begin{bmatrix} 6 & 8 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} 3 \\ 6 \end{bmatrix} = \begin{bmatrix} 8 & 7 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} 3 \\ 6 \end{bmatrix}.$$

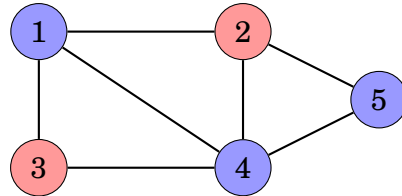
Fakat pratikte algoritmanın hata yapma olasılığı çok küçüktür ve ki biz bu olasılığı $\mathbf{AB} = \mathbf{C}$ olduğunu söylemeden önce sonucu birkaç farklı random \mathbf{X} vektörde deneyerek azaltabiliriz.

Çizge Boyaması

n düğüm ve m kenardan oluşan bir çizgemizin olduğunu ve amacımızın iki renk kullanarak en az $m/2$ kenarın noktalarının farklı renklerine boyandığı bir yol bulmak olsun. Örneğin



çizgesinde geçerli bir boyama aşağıdaki gibidir:



Yukarıdaki çizge 7 kenar içerir ve 5 tanesinde bitiş noktalarının renkleri farklıdır yani bu boyama geçerlidir.

Bu problem bir Las Vegas algoritması ile uygun bir boyama bulana kadar rastgele boyama bularak çözülebilir. Rastgele boyamada her düğümün rengi bağımsız bir şekilde her boyanın olasılığı $1/2$ olacak şekilde çözülür.

Rastgele boyamada, tek bir kenarın bitiş noktalarının farklı renklere sahip olma olasılığı $1/2$ olur. Böylece bitiş noktaları farklı olan kenar sayısının beklenen değeri $m/2$ olur. Böylece rastgele bir boyamanın geçerli olduğu beklendiği için pratikte geçerli bir boyamayı hızlı bir şekilde buluruz.

Bölüm 25

Oyun Teorisi (Game Theory)

Bu bölümde rastgele eleman içermeyen iki kişili oyunlardan bahsedeceğiz. Amacımız rakip ne yaparsa yapsın eğer varsa oyunu kesinlikle kazandıracak bir yol bulmaktır .

Bu tip oyunlar için genel bir strateji vardır ve bu oyunları **nim teorisi** ile analiz edebiliriz. İlk başta oyuncunun öbekten çubuk çıkardığı basit oyunlara bakacağız ve sonra bu oyunlarda kullanılan stratejiyi diğer oyunlara aktaracağız.

25.1 Oyun Durumları

Diyelim ki başta n tane çubuktan oluşan bir öbeğe sahip bir oyun düşünelim. A ve B oyunculari sırasıyla oynar ve A oyuncusu vaşlar. Her harekette oyuncu öbekten 1, 2 veya 3 çubuk çıkarmalıdır ve en son çubuğu çıkaran oyuncu oyunu kazanmaktadır.

Örneğin $n = 10$ olursa oyun şu şekilde gibi ilerleyebilir:

- A oyuncusu 2 çubuk çıkarır (8 çubuk kaldı).
- B oyuncusu 3 çubuk çıkarır (5 çubuk kaldı).
- A oyuncusu 1 çubuk çıkarır (4 çubuk kaldı).
- B oyuncusu 2 çubuk çıkarır (2 çubuk kaldı).
- A oyuncusu 2 çubuk çıkarır ve oyunu kazanır.

Bu oyun her durum sayısının kaç çubuk kaldığını belirten $0, 1, 2, \dots, n$, durumlarından oluşur.

Kazanan ve Kaybeden Durumlar

Kazanan durum oyuncunun oyunu optimal oynaması durumunda kazanacağı durumdur ve **kaybeden durum** da rakip oyunu optimal oynarsa kaybedeceği durumdur. Böylece oyundaki her duruma kazanan veya kaybeden durum olarak sınıflandırabiliriz.

Yukarıdaki oyunda 0. durum kesinlikle kaybeden pozisyonudur çünkü oyuncu burada herhangi bir hamle yapamaz. 1, 2 ve 3. durumlar da kazanan durumlardır çünkü 1, 2 veya 3 çubuk çıkararak oyunu kazanabiliriz. 4. durum da böylece

kaybeden taraftır çünkü yapılacak herhangi bir hareket rakip için kazanacak bir hamle oluşturur.

Genel olarak, eğer şu anki durumdan kaybeden duruma birden fazla yol varsa şu anki durum kazanan durum ve tam tersi durumda da şu anki durum kaybeden durumdur. Bu gözlemi kullanarak oyundaki bütün durumları başka olası hamlenin olmadığı kaybeden durumlardan başlayarak sınıflandırabiliriz.

Yukarıdaki oyunun $0 \dots 15$ durumları aşağıdaki şekilde sınıflandırılabilir (W kazanan durumu L ise kaybeden durumu belirtir):

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L	W	W	W	L	W	W	W	L	W	W	W	L	W	W	W

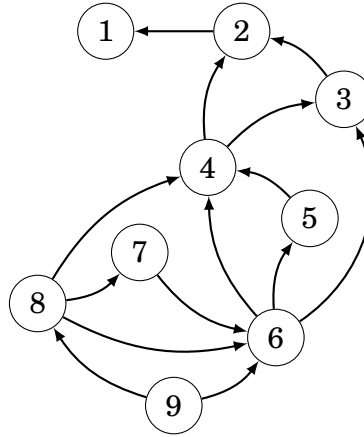
Böylece oyunu analiz etmek kolaydır. Eğer bir k durumunda k 4'e bölünüyorsa k durumu kaybeden bir durumdur aksi taktirde kazanan bir durumdur. Bu oyunu optimal oynamanın yollarından biri her zaman öbekte kalacak çubuk sayısını 4'e bölünebilir miktarda tutarak yapılabilir. En sonda hiç çubuk kalmaz ve rakip kaybetmiş olur.

Tabii bu durum bizim turumuz geldiği zaman kalan çubuk sayısının 4'e bölünemiyor olması gerekir. Eğer öyleyse yapacak bir şeyimiz yoktur ve eğer rakip oyunu optimal oynarsa kazanabilir.

Durum Çizgesi

Şimdi her k durumunda k 'den küçük ve k 'yi bölebilen miktarda x çubuk alabildiğimizi varsayalım. Örneğin 8. durumda 1, 2 veya 4 çukub çıkarabiliriz ama 7. durumda sadece 1 çubuk çıkarabiliriz.

Aşağıdaki resim bir oyunun $1 \dots 9$ durumlarını **durum çizgesinde** gösterir ve burada düğümler durum olur ve kenarlar da arasındaki uygun hareketler olur:



Bu oyunun son durumu her zaman 1. durum olur yani kaybeden bir durum olur çünkü başka hareket edilebilecek durum yoktur. $1 \dots 9$ durumlarının sınıflandırılması aşağıdaki gibi olur:

1	2	3	4	5	6	7	8	9
L	W	L	W	L	W	L	W	L

İlginç bir şekilde bu oyundaki bütün çift sayılı durumlar kazanan durumdur ve bütün tek sayılı durumlar kaybeden durumlardır.

25.2 Nim Oyunu

Nim oyunu oyun teorisinde önemli bir yere sahip olan basit bir oyundur çünkü çoğu oyun aynı taktik kullanılarak oynanabilir. İlk başta, nim üzerinde odaklanırsanız sonrasında stratejiyi diğer oyunlara uygularız.

Nimde n tane öbek vardır ve her öbekte eşit miktarda çubuk vardır. Oyuncular sırasıyla hareket eder ve her turdu oyuncu çubuk içeren bir öbeği seçer ve oradan herhangi bir sayıda çubuk çıkarır. Oyunu kazanan oyuncu ise son çubuğu çıkaran kişidir.

Nimdeki durumlar $[x_1, x_2, \dots, x_n]$ şeklindedir ve burada x_k k öbeğindeki çubuk sayısını verir. Örneğin $[10, 12, 5]$ durumlarında oyunda 10, 12 ve 5 çubuktan oluşan 3 tane öbek olduğunu söyler. $[0, 0, \dots, 0]$ durumu kaybeden bir durumdur çünkü buradan başka hiç düğüm çıkarılamaz ve bu yüzden bu son durumdur.

Analiz

Herhangi bir nim durumunu kolay bir şekilde **nim toplamını** $s = x_1 \oplus x_2 \oplus \dots \oplus x_n$ hesaplayarak sınıflandırabiliriz¹. \oplus xor operatörü anlamına gelir. Nim toplamı 0 olan durumlar kaybeden durum ve geri kalan durumlar kazanan durumdur. Örneğin $[10, 12, 5]$ durumunun nim toplamı $10 \oplus 12 \oplus 5 = 3$ olur ve bu yüzden kazanan durumdur.

Fakat nim toplamı nim oyunuyla nasıl alakalı oluyor? Bunu anlamak için nim durumu değiştiği zaman nim toplamının nasıl değiştiğine bakmamız gerekir.

Kaybeden Durumlar: Son durum $[0, 0, \dots, 0]$ kaybeden durumdur ve beklenildiği gibi nim toplamı 0'dır. Diğer kaybeden durumlarda herhangi bir hareket yapmak kazanan duruma götürür çünkü sadece tek bir x_k değeri değişir ve böylece nim toplamı da değişir yani nim toplamı hareketten sonra 0'dan farklıdır.

Kazanan Durumlar: Eğer $x_k \oplus s < x_k$ olan bir k öbeği varsa kaybeden duruma geçebiliriz. Bu durumda k öbeği $x_k \oplus s$ tane çubuk içerecek şekilde öbekten çubuk çıkarırız. Her zaman öyle bir öbek vardır ki x_k 'nin, s 'in en solundaki bitine karşılık gelen bir biti vardır.

Örneğin $[10, 12, 5]$ durumunu düşünelim. Bu kazanan durumdur çünkü nim toplamı 3'tür. Böylece buradan kaybeden duruma geçilecek bir hareket vardır. Şimdi böyle bir hareket bulacağız.

Durumun nim toplamı aşağıdaki gibidir:

10		1010
12		1100
5		0101
3		0011

¹Nim için optimal strateji 1901 yılında C. L. Bouton tarafından yayınlanmıştır [10],

Bu durumda sadece 10 çubuklu öbek, nim toplamının en solundaki bitine karşılık gelen bite sahiptir:

10	1010
12	1100
5	0101
3	0011

Öbeğin yeni büyüklüğü $10 \oplus 3 = 9$ olmalıdır yani sadece bir çubuk çıkarırız. Bundan sonra durumlar [9, 12, 5] olur ki bu da kaybeden bir durumdur:

9	1001
12	1100
5	0101
0	0000

Misère Oyunu

Bir **misère oyununda** oyunun amacı tam zıttır yani son çubuğu alan oyuncu oyunu kaybeder. Misere nim oyunu da standart nim oyunu gibi optimal şekilde oynanabilir.

Buradaki fikir ilk başta misère oyununu standart oyun gibi oynayıp oyunun sonunda taktiği değiştirmektir. Yeni strateji her öbekte en fazla bir çubuk kaldığı zaman kullanılacaktır.

Standart oyunda, bir çubuğa sahip çift sayıda öbek sayısı bulunmayı sağlayan bir hareket seçmeliyiz. Fakat misère oyununda bir çubuğa sahip tek sayıda öbek bulunmasını sağlayan bir hareket seçmeliyiz.

Bu strateji çalışır çünkü stratejinin değiştiği durum oyunda her zaman bulunur ki bu da kazanan durumdur çünkü bu durum birden fazla çubuk içeren sadece bir tane öbek içerir ve böylece nim toplamı 0 olmaz.

25.3 Sprague–Grundy Teoremi

Sprague–Grundy Teoremi² nim oyunundaki stratejiyi aşağıdaki şartları sağlayan diğer bütün oyunlarda kullanılabilmesini sağlar:

- Sırasıyla turla oyunu oynayan iki oyuncu vardır.
- Oyun durumlardan oluşur ve bir durumdaki olası hareketler kimin sırası olduğuna göre değişmez.
- Oyun bir oyuncu hamle yapamadığı zaman biter.
- Oyun eninde sonunda biter.
- Oyuncular durumlar ve izin verilen hamleler konusunda tam bilgileri vardır ve bu oyunda rastgele bir durum yoktur.

Buradaki fikir her oyun durumu için bir nim öbeğindeki çubuk sayısına karşılık gelecek bir Grundy sayısı hesaplamaktır. Bütün durumların Grundy sayılarını bildiğimiz zaman oyunu nim oyunu gibi oynayabiliriz.

²Bu teoremi, R. Sprague [61] ve P.M Grundy [31], birbirlerinden bağımsız şekilde bulmuşlardır.

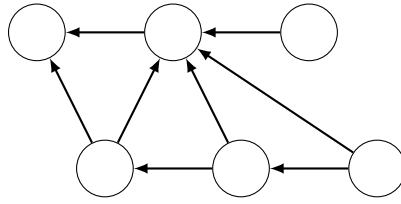
Grundy Sayıları

Bir oyun durumunun **Grundy sayısı**

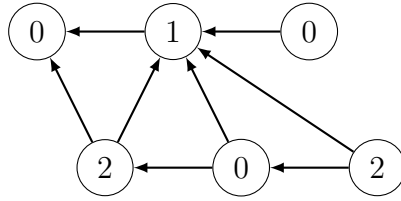
$$\text{mex}(\{g_1, g_2, \dots, g_n\})$$

olur ve burada g_1, g_2, \dots, g_n hareket edebileceğimiz durumların Grundy sayılarıdır. Mex fonksiyonu ise kümede olmayan en küçük negatif olmayan sayıyı verir. Örneğin $\text{mex}(\{0, 1, 3\}) = 2$. Eğer durumda herhangi bir olası hamle yoksa Grundy sayısı 0 olur çünkü $\text{mex}(\emptyset) = 0$.

Örneğin aşağıdaki durum çizgesinde



Grundy sayıları aşağıdaki gibi olur:

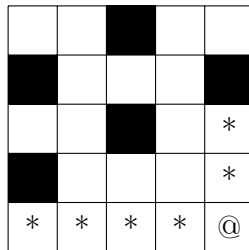


Kaybeden durumun Grundy sayısı 0 olur ve kazanan durumun Grundy sayısı bir pozitif sayı olur.

Bir durumun Grundy sayısı bir nim öbeğindeki çubuk sayısına karşılık gelir. Eğer Grundy sayısı 0 ise Grundy sayısı sadece pozitif olan durumlara giderken eğer Grundy sayısı $x > 0$ ise bütün $0, 1, \dots, x - 1$ Grundy sayılarını içeren durumlara hareket ederiz.

Örneğin, oyuncuların bir labirentte figür hareket ettirdikleri bir oyun düşünelim. Labirentteki her kare ya yer ya da duvardır. Her adımda oyuncu ya belirli bir miktarda sola ya da belirli miktarda yukarıya gitmelidir. Oyunu kazanan oyuncu son hamleyi yapan oyuncudur.

Aşağıdaki resim oyunun olası bir ilk durumunu gösterir ve burada @ figürü ve * ise de hareket edebileceği bir kareyi gösterir.



Oyunun durumları bütün yer kareleri olur. Yukarıdaki labirentte Grundy sayıları şekildeki gibi olur:

0	1		0	1
	0	1	2	
0	2		1	0
	3	0	4	1
0	4	1	3	2

Böylece labirent oyunundaki her durum nim oyunundaki her bir öbeğe karşılık gelir. Örneğin sağ alt karenin Grundy sayısı 2 olur ki bu da kazanan bir durumdur. Kaybeden bir duruma varıp oyunu ya iki adım yukarı giderek ya da dört adım sola giderek kazanabiliriz.

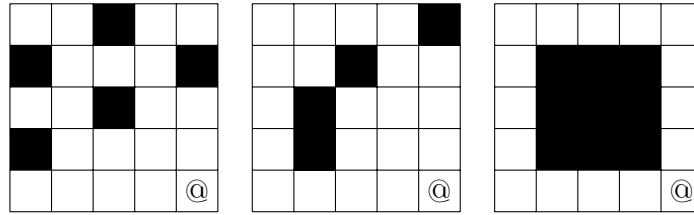
Orijinal nim oyunundakinin aksine Grundy sayısı şu anki olduğumun durumun Grundy sayısından daha büyük bir duruma gidebilmemiz mümkündür. Fakat rakip her zaman bu hamleyi kaldıracak bir hamle seçebilir yani kaybeden durumdan kaçmak mümkün değildir.

Alt Oyunlar

Şimdi oyunumuzun alt oyunlardan oluştuğunu varsayacağız ve her adımda bir oyuncu bir alt oyun seçer ve bu alt oyunda bir hamle yapar. Oyun herhangi bir alt oyunda yapılacak hamle kalmadığında biter.

Bu durumda bir oyunun Grundy sayısı, alt oyunların Grundy sayılarının nim toplamı olur. Bu oyun nim oyunu gibi bütün alt oyunların Grundy sayılarını ve nim toplamlarını hesaplayarak oynamaz mümkündür.

Örnek olarak 3 labirentten oluşan bir oyun düşünelim. Bu oyunda her adımda oyuncu bir labirent seçer ve figürü o labirentte hareket ettirir. Diyelim ki oyunun ilk durumu aşağıdaki gibidir:



Labirentlerin Grundy sayıları aşağıdaki gibi olur:

0	1		0	1
	0	1	2	
0	2		1	0
	3	0	4	1
0	4	1	3	2

0	1	2	3	
1	0		0	1
2		0	1	2
3		1	2	0
4	0	2	5	3

0	1	2	3	4
1				0
2				1
3				2
4	0	1	2	3

İlk durumda Grundy sayılarının nim toplamı $2 \oplus 3 \oplus 3 = 2$ olur yani ilk oyuncu oyunu kazanabilir. Yapılacak optimal bir hareket ilk labirentte iki adım yukarı çıkmaktır ki bu da $0 \oplus 3 \oplus 3 = 0$ nim toplamını oluşturur.

Grundy'nin Oyunu

Bazen oyundaki bir hareket oyunu birbirlerinden bağımsız olacak şekilde alt oyunlara böler. Bu durumda oyunun Grundy sayısı

$$\text{mex}(\{g_1, g_2, \dots, g_n\}),$$

olur ve burada n toplam olası hareket sayısını ve

$$g_k = a_{k,1} \oplus a_{k,2} \oplus \dots \oplus a_{k,m}$$

ve burada k hamlesi $a_{k,1}, a_{k,2}, \dots, a_{k,m}$ Grundy sayılarına sahip alt oyunlar oluşturur.

Bu tip bir oyun **Grundy'nin oyunudur**. Başta n tane çubuktan oluşan tek bir tane öbek vardır. Her adımda oyuncu bir öbeği seçer ve bu öbeği büyüklükleri farklı olacak şekilde iki tane boş olmayan öbeğe böler. Son hamleyi yapan oyuncu oyunu kazanır.

Diyelim ki $f(n)$, n tane çubuk içeren öbeğin Grundy sayısı olsun. Grundy sayısı öbeği iki öbeğe bölecek şekilde yapan bütün yolları deneyerek hesaplayabiliriz. Örneğin $n = 8$ ise olası yollar $1 + 7$, $2 + 6$ ve $3 + 5$ olur yani

$$f(8) = \text{mex}(\{f(1) \oplus f(7), f(2) \oplus f(6), f(3) \oplus f(5)\}).$$

Bu oyunda $f(n)$ değeri $f(1), \dots, f(n-1)$ değerlerine bağlıdır çünkü 1 veya 2 çubuktan oluşan öbekleri bölmek mümkün değildir. İlk Grundy sayıları:

$$\begin{aligned} f(1) &= 0 \\ f(2) &= 0 \\ f(3) &= 1 \\ f(4) &= 0 \\ f(5) &= 2 \\ f(6) &= 1 \\ f(7) &= 0 \\ f(8) &= 2 \end{aligned}$$

$n = 8$ için Grundy sayısı 2 olur yani oyunu kazanmak mümkündür. Kazanan hamle $1 + 7$ şeklinde öbekler oluşturmaktır çünkü $f(1) \oplus f(7) = 0$ olur.

Bölüm 26

Yazı Algoritmaları

Bu bölüm yazı işlemek için kullanılan verimli algoritmaları anlatacak. Çoğu yazı problemi kolay bir şekilde $O(n^2)$ zamanda çözülebilir fakat buradaki amaç istediğimizi $O(n)$ veya $O(n \log n)$ zamanda yapan algoritmalar bulmaktır.

Örneğin temel yazı işleme problemlerinden biri **örüntü bulma** problemidir. Burada n uzunluğunda yazı ve m uzunluğunda örüntü verilir ve amacımız yazıdaki örüntüleri bulmamızdır.

Örüntü bulma problemi kolay bir şekilde $O(nm)$ zamanda kaba kuvvet algoritması ile örüntünün olabileceği bütün pozisyonları deneyerek bulmak mümkündür. Fakat bu bölümde sadece $O(n + m)$ zaman gerektiren daha verimli algoritmalar-
dan bahsedeceğiz.

26.1 Yazı Terminolojisi

Bu bölümde yazılarda sıfır tabanlı indislerin kullanıldığını varsayacağız yani n uzunluğundaki bir s yazısı $s[0], s[1], \dots, s[n-1]$ karakterlerinden oluşur. Örneğin alfabe büyük İngilizce harflerinden $\{A, B, \dots, Z\}$ oluşur (Türkçe karakterler neredeyse hiç kullanılmamaktadır.).

Bir **alt dize** (substring) yazıdaki ardışık birkaç karakterden oluşan bir dizidir. Burada s dizisinde a pozisyonunda başlayıp b pozisyonunda biten bir $s[a \dots b]$ alt dize şeklinde gösterilebilir. n uzunluğundaki bir yazının $n(n+1)/2$ tane alt dizesi vardır. Örneğin ABCD yazısının alt dizeleri A, B, C, D, AB, BC, CD, ABC, BCD ve ABCD olur.

Bir **alt dizi** (subsequence) ardışık olması gerekmeyen karakterlerden olup orijinal pozisyonlarına uygun bir şekilde sıralanmış bir dizidir. n uzunluğundaki bir yazının $2^n - 1$ tane alt dizisi vardır. Örneğin ABCD yazısının alt dizileri A, B, C, D, AB, AC, AD, BC, BD, CD, ABC, ABD, ACD, BCD ve ABCD olur.

Bir **ön ek**, yazının en başında başlayan bir alt dizedir (substring). Bir **son ek** ise yazının en sonunda biten bir alt dizedir. Örneğin ABCD yazısının ön ekleri A, AB, ABC ve ABCD iken son ekleri ise D, CD, BCD ve ABCD olur.

Bir **rotasyon**, yazıdaki bütün karakterleri bir sağa kaydırarak (sondaki harf başa gelir) ya da sola kaydırarak (baştaki harf sona gelir) yapmak mümkündür. Örneğin ABCD yazısının rotasyonları ABCD, BCDA, CDAB ve DABC olur.

Bir **periyot** yazının öyle bir ön ekidir ki yazıyı bu periyodu tekrar ettirerek oluşturabiliriz. Son tekrar bütün periyodu içermeyebilir yani sadece periyodun belirli bir ön ekini içerebilir. Örneğin ABCABCA yazısının en kısa periyodu ABC olur.

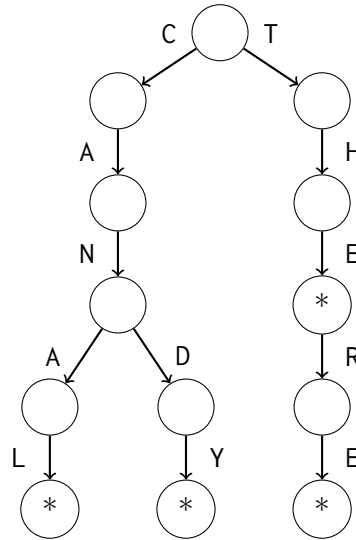
Bir **sınır** (border) yazının hem ön eki hem de son ekidir. Örneğin ABACABA yazısının sınırları A, ABA ve ABACABA olur.

Yazılar **sözlük sıralaması** (lexicographical order) (alfabetik sıralama da denilebilir.) kullanılarak sıralanır. Yani $x < y$ olması için ya $x \neq y$ ve x 'in y 'nin bir ön eki olması veya $i < k$ iken $x[i] = y[i]$ olup $x[k] < y[k]$ durumunu sağlayan bir k pozisyonu olması gerekir.

26.2 Trie Yapısı

Bir **trie**, yazı kümesi tutan bir köklü ağaçtır. Kümedeki her yazı kökten başlayacak şekilde bir karakter zinciriyle tutulur. Eğer iki yazının ortak bir öneki varsa bu yazıların aynı zamanda zincirlerinin belirli bir kısmı ortaktır.

Örneğin aşağıdaki trie'yi düşünelim:



Bu trie {CANAL, CANDY, THE, THERE} kümesini tutar. Düğümdeki * karakteri, kümedeki bir yazının bu düğümde bittiğini belirtir. Böyle bir karakter gereklidir çünkü bir yazı başka bir yazının ön eki olabilir. Örneğin yukarıdaki trie'de THE yazısı THERE yazısının ön ekidir.

Bir trie'nin n uzunluğunda yazı içerip içermediğini $O(n)$ zamanda bulmak mümkündür çünkü kök düğümünden başlayan zinciri takip edebiliriz. Aynı zamanda bir n uzunluğundaki yazıyı trie'ye eklememiz $O(n)$ zamanda olur. Bunu da hali hazırda olan zincirden başlayarak gerekli düğümleri eklememizle olur.

Trie kullanarak, verilen bir yazının kümeye ait en uzun ön ekini bulabiliriz. Hatta her düğümde daha fazla bilgi tutarak verilen ön eke sahip olan kümedeki yazıların sayısını da bulabiliriz.

Trie bir dizide

```
int trie[N][A];
```

N 'in maksimum düğüm sayısını (kümedeki yazıların olabileceği maksimum uzunluk) ve A 'nın alfabenin büyüklüğü olacak şekilde tutabiliriz. Trie'deki düğümler kökün numarası 0 olacak şekilde **0,1,2,...** gibi numaralandırılabilir. `trie[s][c]`, s düğümünden c karakterini kullanarak gittiğimizde zincirde sonraki varacağımızı düğümü gösterir.

26.3 Yazı Hashleme

Yazı hashleme verimli bir şekilde iki yazının aynı olup olmadığını bulmamıza sağlayan bir tekniktir¹. Yazı hashlemedeki fikir yazıların karakterlerini karşılaştırmak yerine hash değerlerini karşılaştırmaktır.

Hash Değerlerini Hesaplamak

Bir yazının **hash değeri** yazıdaki karakterlerden hesaplanan bir sayıdır. Eğer iki yazı aynıysa onların hash değerleri de aynıdır ki böylece yazıları hash değeri ile karşılaştırabiliriz.

Yazı hashleme yapmanın yollarından biri **polinom hashleme**dir. n uzunluğundaki bir s yazısının hash değeri

$$(s[0]A^{n-1} + s[1]A^{n-2} + \dots + s[n-1]A^0) \bmod B$$

şekline olur ki burada $s[0], s[1], \dots, s[n-1]$ s 'in karakterlerinin kodları olarak da yorumlanabilir ve A, B de önceden seçilmiş sabit sayılardır.

Örneğin ALLEY kelimesinin karakterlerinin kodları:

A	L	L	E	Y
65	76	76	69	89

olur. Böylece $A = 3$ ve $B = 97$ ise ALLEY yazısının hash değeri

$$(65 \cdot 3^4 + 76 \cdot 3^3 + 76 \cdot 3^2 + 69 \cdot 3^1 + 89 \cdot 3^0) \bmod 97 = 52.$$

Ön İşleme

Polinom hashlemesi kullanarak bir s yazısının alt dizisinin hash değerini $O(n)$ ön işlemeden sonra $O(1)$ zamanda hesaplayabiliriz. Buradaki fikir $h[k]$ ifadesi, $s[0 \dots k]$ alt dizisinin hash değerini tutacak şekilde bir h dizisi oluşturmaktır. Dizinin değerleri özyinelemeli bir şekilde aşağıdaki gibi hesaplanabilir:

$$\begin{aligned} h[0] &= s[0] \\ h[k] &= (h[k-1]A + s[k]) \bmod B \end{aligned}$$

Bunla beraber $p[k] = A^k \bmod B$ olacak şekilde bir p dizisi de oluşturabiliriz:

$$\begin{aligned} p[0] &= 1 \\ p[k] &= (p[k-1]A) \bmod B. \end{aligned}$$

¹Teknik Karp-Rabin örüntü eşleştirme algoritması ile popülerleşmiştir [42].

Bu dizileri oluşturmak $O(n)$ zaman alır. Bundan sonra herhangi bir $s[a \dots b]$ alt dizisinin hash değerini hesaplamak, $a > 0$ olduğunu varsayarak

$$(h[b] - h[a-1]p[b-a+1]) \bmod B$$

formülünü kullanarak $O(1)$ zaman alır. Eğer $a = 0$ ise hash değeri $h[b]$ olur.

Hash Değerlerini Kullanmak

Hash değerlerini kullanarak verimli bir şekilde yazıları karşılaştırabiliriz. Her yazının karakterlerini teker teker karşılaştırmak yerine hash değerlerini karşılaştırabiliriz. Eğer hash değerleri aynıysa yazıların *muhtemelen* aynıdır ve eğer hash değerleri farklı ise yazılar *kesinlikle* farklıdır.

Hashleme kullanarak bir kaba kuvvet algoritmasını verimli yapabiliriz. Örneğin şu örüntü eşleştirme problemini düşünelim: s yazısı ve p örüntüsü verildiği kabul ederek p 'nin s 'te bulunduğu pozisyonları bulun. Düz kaba kuvvet bir algoritma p 'nin bulunabileceği bütün pozisyonları dener ve yazıları karakter karakter dener. Bu tip bir algoritmanın zaman karmaşıklığı $O(n^2)$ olur.

Kaba kuvvet algoritmayı hashleme sayesinde daha verimli hale getirebiliriz çünkü algoritma yazıların alt dizelerini karşılaştırıyor. Hashleme kullanarak her karşılaştırma sadece $O(1)$ zaman alır çünkü alt dizelerin sadece hash değerleri karşılaştırılır. Böylece bu tip bir algoritmanın zaman karmaşıklığı $O(n)$ olur ki bu da bu problem için en iyi zaman karmaşıklığıdır.

Hashleme ve *ikili aramayı* birleştirerek iki yazının sözlük sırasını logaritmik zamanda bulmak mümkündür. Bunu yazıların ortak ön eklerinin uzunluğunu ikili arama ile bulmak mümkündür. Ortak ön eklerinin uzunluğunu bulduktan sonra ön ekten sonraki karaktere bakmamız yeterlidir çünkü bu yazıların sırasını belirler.

Çarpışmalar (Collisions) ve Parametreler

Hash değerlerini karşılaştırmanın sıkıntılarından biri **çarpışma**dır yani farklı iki yazının aynı hash değerine sahip olma durumudur. Bu durumda algoritma iki yazının aynı olduğunu düşünür fakat gerçekte bunlar aynı değildir ve bu da algoritmanın yanlış cevap vermesine sebep olur.

Çarpışmalar her zaman olasıdır çünkü toplam farklı yazı sayısı toplam farklı hash değeri sayısından daha büyüktür. Fakat A ve B sabitleri dikkatlice seçilirse çarpışma olma ihtimali düşüktür. Bunun bir yola 10^9 'a yakın rastgele sabit sayı seçmektir. Örneğin:

$$\begin{aligned} A &= 911382323 \\ B &= 972663749 \end{aligned}$$

Bu sabit sayıları kullanarak hash değerleri için `long long` türü kullanılabilir çünkü AB ve BB çarpımları `long long` tipine sığabilir fakat 10^9 tane farklı hash değeri bulundurmak yeterli midir?

Hashlemenin kullanılabileceği 3 olası durum düşünelim:

1. *Senaryo*: x ve y yazıları birbirleriyle karşılaştırılır. Çarpışma olmasının ihtimali bütün hash değerlerinin ihtimalinin eşit olması durumunda $1/B$ olur.

2. *Senaryo*: Bir x yazısı y_1, y_2, \dots, y_n yazıları ile karşılaştırılıyor. Bu durumda bir veya daha fazla çarpışma olmasının ihtimali

$$1 - \left(1 - \frac{1}{B}\right)^n.$$

3. *Senaryo*: Bütün x_1, x_2, \dots, x_n yazı çiftleri birbirleriyle karşılaştırılıyor. Bir veya daha fazla kaza olmasının ihtimali

$$1 - \frac{B \cdot (B-1) \cdot (B-2) \cdots (B-n+1)}{B^n}.$$

Aşağıdaki tablo $n = 10^5$ ve B 'nin değeri değiştiği zaman çarpışmaların olasılıklarını gösterir:

sabit B	1. senaryo	2. senaryo	3. senaryo
10^3	0.001000	1.000000	1.000000
10^6	0.000001	0.632121	1.000000
10^9	0.000000	0.001000	1.000000
10^{12}	0.000000	0.000000	0.393469
10^{15}	0.000000	0.000000	0.000500
10^{18}	0.000000	0.000000	0.000001

Tablo 1. senaryodaki çarpışma ihtimalinin $B \approx 10^9$ olduğu zaman ihmal edilebilir olduğunu, 2. senaryoda çarpışma ihtimalinin olası olduğunu ama yine de çok küçük olduğunu fakat 3. senaryoda durumun çok farklı olduğunu söyler. Eğer $B \approx 10^9$ ise bir çarpışma yüksek ihtimal olacaktır.

3. senaryodaki durum aynı zamanda **doğum günü paradoksu** olarak da bilinir: Eğer bir odada n tane insan varsa, odadaki *rastgele* iki kişinin aynı doğum gününe sahip olması n çok küçük olsa dahi yüksektir. Hashlemede o zaman bütün değerler karşılaştırıldığı zaman iki tane hash değerinin aynı olma ihtimali yüksektir.

Çarpışma olasılığını küçültmek için farklı parametreler kullanarak *birkaç* tane hash değeri hesaplayabiliriz. Aynı anda bir çarpışmanın bütün hash değerlerinde olma ihtimali yok denecek kadar azdır. Örneğin $B \approx 10^9$ parametresine sahip iki hash değeri bir $B \approx 10^{18}$ parametresine sahip bir hash değerine karşılık gelir ki bu durumda çarpışmanın olması çok düşüktür.

Bazı insanlar rahat olduğu için $B = 2^{32}$ ve $B = 2^{64}$ sabitlerini kullanırlar çünkü 32 ve 64 bit tam sayılarla olan işlemler 2^{32} ve 2^{64} sayılarının modulosu ile hesaplanır. Fakat bu iyi bir seçenek **değildir** çünkü 2^x formundaki bir sabit kullanıldığı zaman her zaman çarpışma oluşturan girdi oluşturmak mümkündür [51].

26.4 Z-Algoritması

Bir n uzunluğundaki s yazısının z **Z-dizisi**, her $k = 0, 1, \dots, n-1$ için k pozisyonunda başlayan s 'nin ön eki olan en uzun alt dizeyi tutar. Böylece $z[k] = p$ bize $s[0 \dots p-1]$ ile $s[k \dots k+p-1]$ ifadelerinin eşit olduğunu söyler. Çoğu yazı işleme algoritması verimli bir şekilde Z-dizisi kullanılarak çözülebilir.

Örneğin ACBACDACBACBACDA yazısının Z-dizisi aşağıdaki gibidir:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
–	0	0	2	0	0	5	0	0	7	0	0	2	0	0	1

Bu durumda örneğin $z[6] = 5$ olur çünkü 5 uzunluğundaki ACBAC alt dizesi s yazısının ön ekidir ama 6 uzunluğundaki ACBACB alt dizesi s yazısının ön eki değildir.

Algoritmanın Açıklaması

Şimdi **Z-algoritması**² anlatacağız. Bu algoritma Z-dizisini verimli bir şekilde $O(n)$ zamanda oluşturur. Algoritma Z-dizisinin değerlerini soldan sağa hem kaydedilmiş Z-dizisindeki bilgilerle hem de alt dizeleri karakter karakter karşılaştırarak buluruz.

Z-dizisinin değerlerini verimli bir şekilde bulmak için algoritma bir $[x, y]$ aralığı tutar ki $s[x \dots y]$ s yazısının bir ön eki olur ve y olabildiği kadar büyük olur. $s[0 \dots y - x]$ ve $s[x \dots y]$ ifadeleri eşit olduklarını bildiğimiz için bu bilgiyi kullanarak $x + 1, x + 2, \dots, y$ pozisyonlarındaki Z-değerlerini hesaplayabiliriz.

Her k pozisyonunda ilk başta $z[k - x]$ değerini karşılaştırırız. Eğer $k + z[k - x] < y$ ise $z[k] = z[k - x]$ olduğunu biliriz. Fakat $k + z[k - x] \geq y$ ise $s[0 \dots y - k]$ ve $s[k \dots y]$ birbirlerine eşittir ve $z[k]$ değerini bulmak için alt dizeleri karakter karakter karşılaştırmamız gerekir. Yine de algoritma $O(n)$ zamanda çalışır çünkü karşılaştırmaya $y - k + 1$ ve $y + 1$ pozisyonlarında başlarız.

Örneğin aşağıdaki Z-dizisini inşaa edelim:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
–	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

$z[6] = 5$ değerini hesapladıktan şu anki $[x, y]$ aralığı $[6, 10]$ olur:

						$\begin{array}{c} x \qquad \qquad \qquad y \\ \hline \end{array}$									
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
–	0	0	2	0	0	5	?	?	?	?	?	?	?	?	?


Şimdi Z-dizisinin değerlerini verimli bir şekilde hesaplayabiliriz çünkü $s[0 \dots 4]$ ve $s[6 \dots 10]$ değerlerinin aynı olduğunu biliyoruz. İlk başta $z[1] = z[2] = 0$ olduğu için $z[7] = z[8] = 0$ olduğunu da biliriz:

²Z-algoritması [32] kaynağında linear zamanda örüntü bulmak için en kolay algoritma olduğunu verir ve orijinal fikir [50] kaynağından gelir.

						x					y				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
—	0	0	2	0	0	5	0	0	?	?	?	?	?	?	?

Sonra $z[3] = 2$ olduğu için $z[9] \geq 2$ olduğunu da biliriz:

						x					y				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
—	0	0	2	0	0	5	0	0	?	?	?	?	?	?	?



Fakat, 10. pozisyonundan sonra yazı hakkında herhangi bir bilginiz yok ve bu yüzden alt dizeleri karakter karakter karşılaştırmamız gerekir:

						x					y				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
—	0	0	2	0	0	5	0	0	?	?	?	?	?	?	?

$z[9] = 7$ olduğunu görüyoruz yani yeni $[x, y]$ aralığı $[9, 15]$ olur:

									x						y					
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15					
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A					
—	0	0	2	0	0	5	0	0	7	?	?	?	?	?	?					

Bundan sonra geriye kalan bütün Z-dizisi değerleri şu an Z-dizisinde bulunan bilgilerle bulunabilir:

									x					y				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15			
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A			
—	0	0	2	0	0	5	0	0	7	0	0	2	0	0	1			

Z-Dizisini Kullanmak

Z-algoritmasını ya da yazı hashlemesini kullanmak biraz kişiye bağlıdır. Z-algoirtmasının yararı hashlemede olan çarpışmaları içermeyip her zaman çalışmasıdır. Fakat Z-algoritmasını koda geçirmek daha zordur ve bazı problemler sadece hashleme ile çözülebilir.

Örneğin örüntü bulma problemini yeniden düşünelim. Burada amacımız s yazısında p örüntüsünün bulunduğu yerleri bulmaktır. Bu problemi yazı hashleme ile kullandık fakat Z-algoritması bu problem için farklı bir yol verir.

Yazı işlemeindeki genel fikir, özel karakterlerle ayrılmış birkaç yazıdan oluşan bir yazı oluşturmaktır. Bu problemde $p\#s$ yazısını inşaa edebiliriz ki burada p ve s yazıları hiçbir yazıda bulunmayan özel $\#$ karakteriyle ayrılmışlardır. $p\#s$ 'nin Z-dizisi bize p 'nin s 'te bulunduğu pozisyonları verir çünkü bu tip pozisyonlar p uzunluğunu içerir.

Örneğin $s = \text{HATTIVATTI}$ ve $p = \text{ATT}$ ise Z-dizisi aşağıdaki gibi olur:

0	1	2	3	4	5	6	7	8	9	10	11	12	13
A	T	T	#	H	A	T	T	I	V	A	T	T	I
-	0	0	0	0	3	0	0	0	0	3	0	0	0

5 ve 10. pozisyonlar 3 değerini içerir yani ATT örüntüsü HATTIVATTI yazısında bu pozisyonlarda bulunduğu söyler.

Bu algoritmanın zaman karmaşıklığı lineardir çünkü Z-dizisini oluşturup değerlerden geçmek yeterlidir.

Implementasyon

Aşağıdaki kod, Z-dizisine karşılık gelen bir vektör döndüren Z-algoritmasının kodudur.

```
vector<int> z(string s) {
    int n = s.size();
    vector<int> z(n);
    int x = 0, y = 0;
    for (int i = 1; i < n; i++) {
        z[i] = max(0, min(z[i-x], y-i+1));
        while (i+z[i] < n && s[z[i]] == s[i+z[i]]) {
            x = i; y = i+z[i]; z[i]++;
        }
    }
    return z;
}
```

Bölüm 27

Karekök Algoritmaları

Karekök algoritmaları zaman karmaşıklığı karekök olan algoritmalarlardır. Karekök "Fakir adamın logaritması" (poor man's logarithm) olarak görülebilir çünkü: $O(\sqrt{n})$ zaman karmaşıklığı $O(n)$ zaman karmaşıklığından daha iyi ama $O(\log n)$ zaman karmaşıklığından daha kötüdür. Yine de pratikte karekök algoritmaları hızlı ve işe yarar. Örnek olarak bir dizi üzerinde aşağıdaki bu iki işlemi yapabilen bir veri yapısı (data structure) oluşturalım: istenen pozisyondaki elemanı değiştirmek ve dizinin bir aralığındaki elemanların toplamını almak. Önceki bölümlerde, bu problemi her işlemi $O(\log n)$ zamanda yapabilen binary indexed ve segment tree ile çözülebileceğini anlatmıştık. Fakat, biz bu soruyu eleman değiştirmeyi $O(1)$ zamanda ve toplam hesaplamayı $O(\sqrt{n})$ zamanda yapan bir karekök yapısı kullanarak çözeceğiz. Buradaki fikir, diziyi büyüklüğü \sqrt{n} olan *bloklara* ayırarak yapabiliriz. Her blokta o blokta bulunan elemanların toplamı bulunacak. Örneğin 16 eleman büyüklüğündeki bir diziyi aşağıdaki gibi 4 elemanlı bloklara ayırabiliriz:

21				17				20				13			
5	8	6	3	2	7	2	6	7	1	7	5	6	2	3	2

Bu yapıda, dizi elemanlarını değiştirmek kolaydır çünkü her değiştirmeden sonra sadece tek bir bloğun toplamını güncellememiz gerekir. O yüzden bu işlem $O(1)$ sürede yapılabilir. Örneğin aşağıdaki resim elemanın değeri ile bulunduğu bloktaki toplamın nasıl değiştiğini gösteriyor:

21				15				20				13			
5	8	6	3	2	5	2	6	7	1	7	5	6	2	3	2

Sonra, belirli bir aralıktaki toplamı almak için bu aralığı öyle bir 3 parçaya bölmeliyiz ki toplam, sınırlardaki tek elemanların değerleri ile aralarındaki blokların toplamlarından oluşmalıdır:

21				15				20				13			
5	8	6	3	2	5	2	6	7	1	7	5	6	2	3	2

Tek eleman sayısı da ve toplam blok sayısı da $O(\sqrt{n})$ olduğu için aralığın toplamını alma işlemi de $O(\sqrt{n})$ zaman alır. Blok büyüklüklerini $O(\sqrt{n})$ yapmanın amacı iki şeyi *dengelemesinden* dolayıdır: Dizi \sqrt{n} tane bloka bölünür ve her blok \sqrt{n} eleman içerir. Pratikte \sqrt{n} 'in tam değerini parametre olarak kullanılmasına gerek yoktur ve bunun yerine k ve n/k parametreleri kullanılabilir. Burada k , \sqrt{n} 'den farklıdır. En iyi parametre soruya ve girdiye göre değişir. Örneğin eğer algoritma genel olarak bloklarından geçip tekli elemanlardan nadiren geçiyorsa diziyi her biri $n/k > \sqrt{n}$ eleman içeren $k < \sqrt{n}$ bloka bölmek daha mantıklı olacaktır.

27.1 Algoritmaları Birleştirme (Combining Algorithms)

Bu bölümde iki algoritmayı tek algoritmaya birleştiren karekök algoritmalarından bahsedeceğiz. Bu algoritmaları birleştirmeden ayrı ayrı kullandığımızda problemi $O(n^2)$ zamanda çözer ama eğer algoritmaları birleştirirsek problemi $O(n\sqrt{n})$ zamanda çözebiliriz.

Durum İnceleme (Case Processing)

Diyelim n hücreden oluşan iki boyutlu bir yapıya sahip olduğumuzu düşünelim. Her hücrede bir harf var ve amacımız aynı harfe sahip aralarındaki mesafe en kısa olan iki tane hücre bulmaktır. (x_1, y_1) ve (x_2, y_2) hücrelerin arasındaki mesafe $|x_1 - x_2| + |y_1 - y_2|$ olur. Örneğin aşağıdaki yapıya bakalım:

A	F	B	A
C	E	G	E
B	D	A	F
A	C	B	D

Bu durumda en kısa mesafe 2 olup iki 'E' harfi arasındadır. Bu problemi her harfi ayrı ayrı düşünerek çözebiliriz. Bu yöntemi kullanarak oluşan yeni problem spesifik bir *harfe* sahip olan hücreler arasında en kısa mesafeye sahip hücrelerin mesafesini bulmak. Bu problem için aşağıdaki iki algoritmayı kullanırız:

1. *Algoritma*: c harfine sahip bütün hücre çiftlerini dolaşıp aralarındaki en kısa mesafeyi bulmak. Bu algoritma c harfine sahip k hücre için $O(k^2)$ zaman harcar.

2. *Algoritma*: c harfine sahip olan hücrelerin hepsinde aynı anda başlayan bir genişlik öncelikli arama yapmak. c harfine sahip iki hücre arasındaki en kısa mesafe $O(n)$ zamanda bulunur. Bu soruyu çözenin yollarından biri bu iki algoritmadan birisini her harf için kullanmaktır. Eğer sadece 1. algoritmayı kullanırsak zaman karmaşıklığı $O(n^2)$ olur çünkü en kötü durumda her hücre aynı harfe sahip olur ve bu durumda $k = n$ olur. Sadece 2. algoritmayı kullanırsak zaman

karmaşıklığı yine $O(n^2)$ olur çünkü en kötü durumda her hücre farklı harfe sahip olur ve bu durumda n arama yapılması gerekir.

Ama bu iki algoritmayı *birleştirip* her harfin yapıda bulunma miktarına göre farklı algoritma kullanabiliriz. Diyelim ki c harfi k defa bulunuyor. Eğer $k \leq \sqrt{n}$ ise 1. algoritmayı ve eğer $k > \sqrt{n}$ ise 2. algoritmayı kullanırız. Bunu yaparak problemi toplam $O(n\sqrt{n})$ zamanda çözebiliriz.

İlk başta bir c harfi için 1. algoritmayı kullandığımızı varsayalım. c harfi en fazla \sqrt{n} miktarda bulunduğu için c harfine sahip diğer hücrelerle $O(\sqrt{n})$ defa karşılaştırırız. Böylece bu tip hücreleri işleme zamanı $O(n\sqrt{n})$ olur. Sonra başka bir c harfi için 2. algoritmayı kullandığımızı varsayalım. Bu tip harf sayısı \sqrt{n} olduğu için bunları işleme $O(n\sqrt{n})$ zaman alır.

Grup İşleme (Batch processing)

Sonraki problemimizde yine n tane hücreden oluşan iki boyutlu yapımız var. Başta bir tane hücre hariç geriye kalan bütün hücreler beyazdır. Her birinde bir beyaz hücreden siyah hücreye en kısa mesafeyi bulup sonrasında beyaz hücreyi siyaha boyadığımız $n - 1$ işlem yaparız. Örneğin aşağıdaki işleme bakın:

		*	

İlk başta * ile işaretli beyaz hücreden bir siyah hücreye olan en kısa mesafeyi hesaplarız. Bu durumda en kısa mesafe 2'dir çünkü iki adım sol yaparak bir siyah hücreye varabiliriz. Sonrasında bu beyaz hücreyi siyaha boyarız.

Aşağıdaki iki algoritmayı düşünün:

1. *Algoritma:* Genişlik öncelikli aramayı (BFS) kullanarak her beyaz hücre için en yakın siyah hücreye olan mesafeyi buluruz. Bu $O(n)$ zamanda yapılır ve aramadan sonra herhangi bir beyaz hücreden bir siyah hücreye en kısa mesafeyi $O(1)$ sürede buluruz.

2. *Algoritma:* Siyaha boyanmış hücreleri bir listede tutun. Her işlemde bu listeden geçip listeye yeni bir hücre ekleyin. İşlem k uzunluğunda bir liste için $O(k)$ zaman alır. Yukarıdaki algoritmaları işlemleri $O(\sqrt{n})$ gruba bölerek birleştirebiliriz. Böylece her grupta $O(\sqrt{n})$ tane işlem vardır. Her grubun başında 1. Algoritmayı uyguluyoruz ve sonrasında gruptaki işlemleri işlemek için 2. Algoritmayı kullanırız. 2. Algoritmanın listesini gruptan gruba geçerken temizleriz. Her

operasyonda bir siyah hücreye olan en kısa mesafe ya 1. Algoritma ile ya da 2. Algoritma ile hesaplanır.

Bu algoritmaları birleştirilerek oluşan algoritma $O(n\sqrt{n})$ zamanda çalışır. 1. Algoritma $O(\sqrt{n})$ defa uygulanır ve her arama $O(n)$ zaman alır. 2. Algoritmayı grup içinde kullandığı zaman liste $O(\sqrt{n})$ tane hücreden oluşur çünkü gruplar arası geçiş yaparken listeyi temizleriz ve her işlem $O(\sqrt{n})$ zaman alır.

27.2 Sayı Bölmesi (Integer Partitions)

Eğer bir pozitif tamsayı n , pozitif tamsayı toplamı şeklinde gösterilebiliyorsa bu toplam en fazla $O(\sqrt{n})$ tane farklı pozitif tamsayı içerir. Bazı karekök algoritmaları bu gözleme göre yapılmıştır. Bu gözlemin doğru olmasının nedeni ise farklı en fazla tamsayı almak için *küçük* sayıları seçeriz. Eğer $1, 2, \dots, k$ sayılarını alırsak toplam

$$\frac{k(k+1)}{2}.$$

olur. Bu yüzden toplam alınabilecek en fazla farklı sayı miktarı $k = O(\sqrt{n})$ olur. Şimdi, bu gözlemi kullanarak çözülebilen iki sorudan bahsedeceğiz.

Knapsack

Bize toplamı n olan tamsayı ağırlıklarından oluşmuş bir liste verildiğini varsayalım. Amacımız bu ağırlıkları kullanarak oluşturulabilecek bütün toplamı bulmak. Örnek eğer ağırlıklarımız $\{1, 3, 3\}$ ise oluşturulabilecek toplamlar:

- 0 (empty set)
- 1
- 3
- $1 + 3 = 4$
- $3 + 3 = 6$
- $1 + 3 + 3 = 7$

olur. Klasik knapsack yaklaşımı (Bölüm 7.4) kullanılarak bu problemi şu şekilde çözebiliriz: $\text{possible}(x, k)$ adında eğer x toplamı, ilk k ağırlık kullanılarak oluşturulabiliyorsa 1, oluşturulamıyorsa da 0 verecek bir fonksiyon tanımlarız. Bütün ağırlıkların toplamı n olduğu için en fazla n tane ağırlık vardır ve dinamik programlama kullanılarak fonksiyonun bütün değerleri $O(n^2)$ zamanda hesaplanabilir. Fakat, bu algoritmayı en fazla $O(\sqrt{n})$ tane *farklı* ağırlık olduğunu bilerek daha verimli hale getirebiliriz. Böylece ağırlıkları benzer ağırlıklara sahip gruplarda işleyebiliriz. Her grubu $O(n)$ zamanda işleriz ve böylece bu bize $O(n\sqrt{n})$ zaman karmaşıklığına sahip bir algoritma verir. Buradaki fikir, ağırlıkların toplamını tutan bir dizi kullanmaktır. Bu dizi şu ana kadar işlenmiş grupları kullanarak oluşturulabilir. Dizi n eleman içerir: Eğer toplam k oluşturulabiliyorsa 1 oluşturulamıyorsa 0 olur. Grupları işlemek için diziyi soldan sağa gezip şimdiki ve önceki gruplarla oluşturulabilecek yeni ağırlık toplamlarını kaydederiz.

Yazı Oluşturmak (String Construction)

n uzunluğundaki bir s yazısı ile toplam uzunluğu m olan D yazıları verildiğini düşünelim. Bizden istenen D yazılarını birleştirerek kaç farklı şekilde s yazısını oluşturabileceğimizi bulmak. Örneğin, eğer $s = ABAB$ ve $D = \{A, B, AB\}$ ise 4 farklı yolla oluşturabiliriz:

- $A + B + A + B$
- $AB + A + B$
- $A + B + AB$
- $AB + AB$

Bu problemi dinamik programlama kullanarak çözebiliriz: $\text{count}(k)$, D 'de bulunan yazıları kullanarak $s[0 \dots k]$ kısmını kaç tane yolla oluşturulabildiğini veren bir fonksiyon olduğunu düşünelim. Şimdi $\text{count}(n-1)$ bize problemin cevabını verir ve biz problemi önek ağacı (trie) yapısını kullanarak $O(n^2)$ sürede çözebiliriz. Fakat bu soruyu yazı karmaşasıyla (string hashing) ve D 'de en fazla $O(\sqrt{m})$ tane farklı yazı uzunluğu bulunduğunu bilmemizle daha verimli şekilde çözebiliriz. İlk başta, D 'nin bütün karım değerlerini tutan bir H takımı oluştururuz. Sonra bir $\text{count}(k)$ değerini hesaplarken D 'de bulunan p uzunluğunda olan bir yazı olduğunu bilerek bütün p değerlerinden geçiyoruz ve sonrasında $s[k-p+1 \dots k]$ ifadesinin karım değerini hesaplayıp H 'e ait olup olmadığına bakarız. En fazla $O(\sqrt{m})$ tane farklı yazı uzunluğu olduğu için $O(n\sqrt{m})$ sürede çalışan bir algoritma oluşturmuş oluruz.

27.3 Mo'nun Algoritması (Mo's Algorithm)

Mo'nun Algoritması¹ *sabit* diziler (dizi değerleri sorgularla değişmediği) üzerinde yapılan aralık sorgularında kullanılmaktadır. Her sorguda bir $[a, b]$ aralığı verilir ve bizim a ve b pozisyonları arasındaki dizi elemanlarından bir değer bulmamız istenir. Dizi sabit olduğu için sorgular herhangi bir sırada işlenebilir ve Mo'nun Algoritması da bu sorguları özel bir sırada işleyerek algoritmanın verimli olmasını garantiler. Mo'nun Algoritması dizi üzerinde *aktif bir aralık* tutar ve aktif bölgeyi kapsayan sorguya cevap her an bilinebilir. Algoritma sorguları teker teker işler ve her zaman aktif bölge sınırlarını eleman ekleyerek veya çıkararak değiştirir. Bu algoritmanın zaman karmaşıklığı n tane sorgu ve eleman çıkarma ekleme işleminin $O(f(n))$ zaman aldığı durumda $O(n\sqrt{n}f(n))$ olur. Mo'nun Algoritması taktik, sorguların işleme sırasıdır. Dizi $k = O(\sqrt{n})$ 'lik bloklara ayrılır ve $[a_1, b_1]$ sorgusu $[a_2, b_2]$ sorgusundan

- $\lfloor a_1/k \rfloor < \lfloor a_2/k \rfloor$ veya
- $\lfloor a_1/k \rfloor = \lfloor a_2/k \rfloor$ ve $b_1 < b_2$.

¹[12] kaynağına göre bu algoritmanın ismi Mo Tao isimli bir Çinli rekabetçi programcıdan geldiğini söylüyor fakat bu teknik literatürde daha eskiden [44] kaynağında çıkmıştır.

olması durumunda daha önce işlenir. Böylece, sol bitiş noktaları aynı blokta olanlar sağ bitiş noktalarına göre sıralanıp işlenirler. Algoritma bu sırayı kullanarak sadece $O(n\sqrt{n})$ tane işlem yapar çünkü sol bitiş noktası $O(n\sqrt{n})$ adımda $O(n)$ defa hareket eder ve sağ bitiş noktası da $O(n)$ adımda $O(\sqrt{n})$ defa hareket eder. Bu yüzden her iki bitiş noktası da algoritmanın çalışması sırasında toplam $O(n\sqrt{n})$ defa hareket eder.

Örnek

Örneğin, bize her biri dizi içinde bir aralığa tekabül eden bir grup sorgu verildiğini düşünelim. Amacımız her sorguda verilen aralıkta kaç tane *farklı* eleman olduğunu bulmak. Mo'nun Algoritmasında sorgular hep aynı yolla sıralanır ama yine de sorgunun cevabının nasıl tutulması gerektiği probleme bağlıdır. Bu problemde `count` isimli bir dizi tutabiliriz. `count[x]` x elemanının aktif aralıkta kaç defa bulunduğunu gösterir. Sorgudan sorguya geçtiğimiz zaman aktif aralık değişir. Örneğin eğer şu anki aralık:

4	2	5	4	2	4	3	3	4
---	---	---	---	---	---	---	---	---

sonraki aralık:

4	2	5	4	2	4	3	3	4
---	---	---	---	---	---	---	---	---

ise üç adım olacaktır: sol bitiş noktası 1 defa sağa ve sağ bitiş noktası 2 defa sağa kayacaktır. Her adımda, `count` dizisinin güncellenmesi gerekir. x elemanı eklendikten sonra `count[x]` değerini 1 arttırırız ve eğer bu işlemden sonra `count[x] = 1` oluyorsa sorgu cevabını 1 arttırırız. Benzer şekilde x elemanını çıkardığımız zaman `count[x]` değerini 1 azaltırız ve eğer bu işlemden sonra `count[x] = 0` oluyorsa sorgunun cevabını 1 azaltırız.

Bu problemde her adımı yapmak için gereken zaman $O(1)$ olur ve algoritmanın toplam zaman karmaşıklığı $O(n\sqrt{n})$ olur.

Bölüm 28

Segment Ağacının Devamı

Segment ağacı bir sürü algoritma probleminde kullanabilecek olan çok yönlü bir veri yapısıdır. Fakat, segment ağacı hakkında daha anlatmadığımız bir sürü konu var. Şimdi segment ağacının daha gelişmiş konularından bahsedelim.

Şimdiye kadar segment ağacı üzerinde *aşağıdan yukarıya* gidecek şekilde operasyonlar tanımladık. Örneğin aralık toplamlarını Bölüm 9.3'te şekildeki gibi hesaplamıştık:

```
int sum(int a, int b) {
    a += n; b += n;
    int s = 0;
    while (a <= b) {
        if (a%2 == 1) s += tree[a++];
        if (b%2 == 0) s += tree[b--];
        a /= 2; b /= 2;
    }
    return s;
}
```

Fakat daha gelişmiş segment ağaçlarında operasyonları genel olarak *yukarıdan aşağıya* doğru yaparız. Bu yaklaşımı kullanarak fonksiyon aşağıdaki gibi olur:

```
int sum(int a, int b, int k, int x, int y) {
    if (b < x || a > y) return 0;
    if (a <= x && y <= b) return tree[k];
    int d = (x+y)/2;
    return sum(a,b,2*k,x,d) + sum(a,b,2*k+1,d+1,y);
}
```

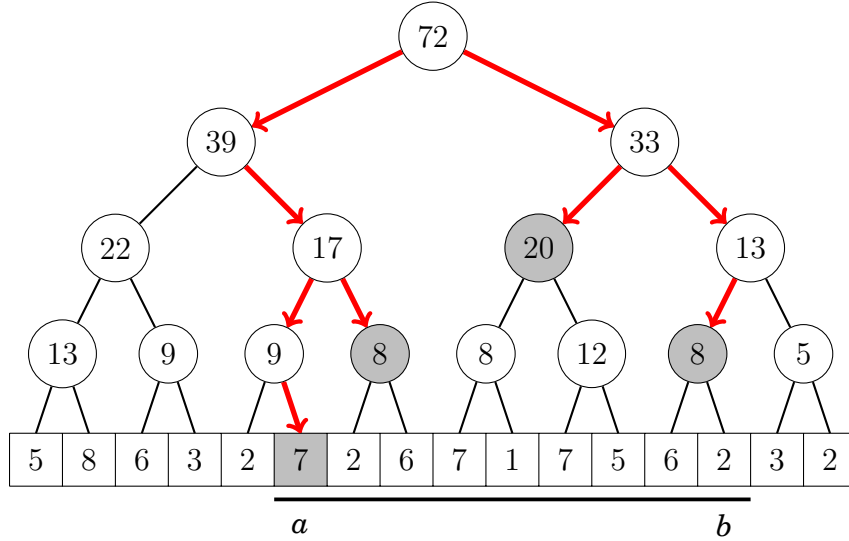
Şimdi herhangi bir $\text{sum}_q(a,b)$ değerini $[a,b]$ aralığındaki dizi değerlerinin toplamı) aşağıdaki gibi hesaplayabiliriz:

```
int s = sum(a, b, 1, 0, n-1);
```

k parametresi ağaçtaki şu an bulunduğumuz pozisyonu gösterir. Başta k , 1'e

eşit olursa çünkü ağacın kökünden başlarız. $[x, y]$ aralığı k 'ye karşılık gelir ve ilk başta $[0, n - 1]$ olur. Toplamı hesaplarken eğer $[x, y]$ aralığı $[a, b]$ aralığının dışındaysa toplam 0'dır. Eğer $[x, y]$ aralığı tamamıyla $[a, b]$ aralığının içindeyse toplam ağaçta bulunabilir. Eğer $[x, y]$ aralığı kısmi olarak $[a, b]$ aralığının içindeyse arama özyinelemeli bir şekilde $[x, y]$ aralığının sol ve sağ taraflarında devam eder. Sol yarım $[x, d]$ ise sağ yarım da $[d + 1, y]$ olur ve burada $d = \lfloor \frac{x+y}{2} \rfloor$.

Aşağıdaki resim $\text{sum}_q(a, b)$ değerini hesaplarken aramanın nasıl devam ettiğini gösterir. Gri düğümler özyinelemenin bittiği düğümleri gösterir ve toplam ağaçta bulunur.



Bu implementasyonda da operasyonlar $O(\log n)$ zaman alır çünkü toplam ziyaret edilen düğümler $O(\log n)$ olur.

28.1 Lazy propagation

Lazy propagation kullanarak hem aralık güncellemelerini hem de aralık sorguları $O(\log n)$ zamanda yapılabilir. Buradaki fikir yukarıdan aşağıya güncellemeler ve sorgular uygulayabilmektedir ve güncellemeleri *tembel* bir şekilde yapmaktır ki böylece sadece gerekli olduğu zaman ağacın aşağısına ilerlenir.

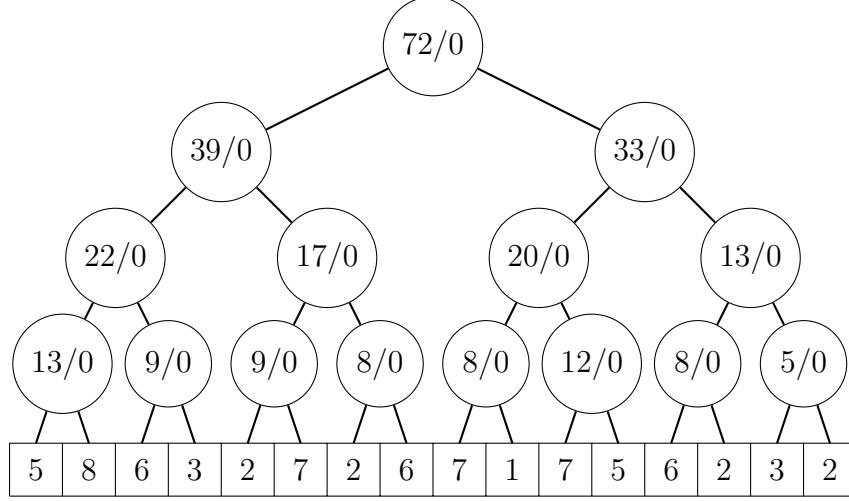
Tembel segment ağacında düğümler iki bilgi içerir. Klasik segment ağacındaki gibi her düğüm toplam veya alt diziyile alakalı bir bilgi içerir. Bununla beraber düğüm ekstradan çocuklarına gönderilmemiş tembel güncellemelerle alakalı bilgi de içerilebilir.

İki tip aralık güncellemesi vardır: aralıktaki her dizi değeri ya belirli miktarda *arttırılır* ya da belirli bir miktara *güncellenir*. Her iki operasyon da benzer fikirlerle koda geçirilebilir ve iki operasyonu da aynı zamanda yapabiliriz.

Tembel Segment Ağaçları (Lazy Segment Trees)

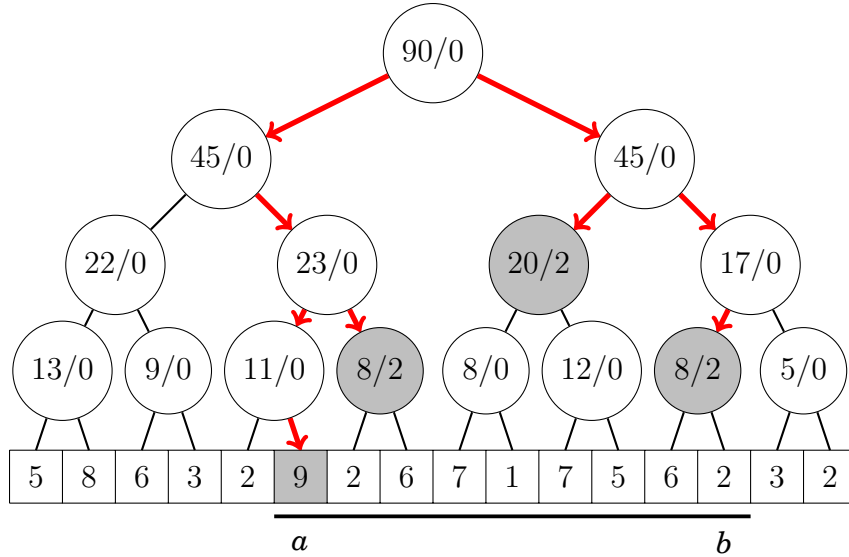
Diyelim ki amacımız iki operasyonu yapabilen bir segment ağacı yapmak olsun: $[a, b]$ aralığındaki her değeri ya bir sabit sayıyla arttırmak veya $[a, b]$ aralığındaki değerlerine toplamını bulmak.

Her düğüm s/z diye iki değer tuttuğu bir ağaç yapalım: s aralıktaki değerlerin toplamını ve z de tembel güncellemedeki değeri yani aralıktaki her değer z kadar arttırılacağını söyler. Aşağıdaki ağaçta her düğüm $z = 0$ ise var olan herhangi bir tembel güncelleme olmadığını gösterir.



$[a, b]$ aralığındaki elemanlar u arttırıldığı zaman kökten yapraklara yürürüz ve düğümdeki ağaçları aşağıdaki gibi değiştiririz: Eğer bir düğümün $[x, y]$ aralığı tamamiyle $[a, b]$ aralığının içindeyse düğümün z değerini u kadar arttırıp durdururuz. Eğer $[x, y]$ kısmı olarak $[a, b]$ 'ye aitse o zaman düğümün s değerini hu kadar arttırılmalıyız ve burada h , $[a, b]$ ile $[x, y]$ 'nin kesişim büyüklüğü (ortak eleman sayısı) olur ve ağaçta özyinelemeli bir şekilde yürümeye devam ederiz.

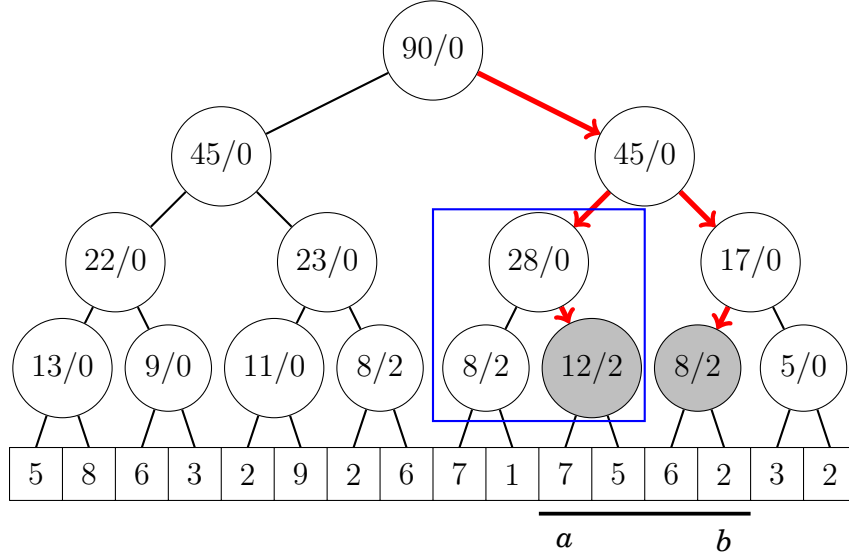
Örneğin, aşağıdaki resim, $[a, b]$ 'deki değerleri 2 arttırdığımız zaman ağacı gösterir:



$[a, b]$ aralığındaki değerlerin toplamını hesaplamak için yukarıdan aşağıya doğru yürürüz. Eğer bir düğümün $[x, y]$ aralığı tamamiyle $[a, b]$ 'ye aitse düğümün s değerini toplama ekleriz. Yoksa aramaya ağacın aşağısında özyinelemeli bir şekilde devam ederiz.

Hem güncellemelerde ve sorgularda, tembel güncellemenin değeri her zaman düğümü işlemeyen önce çocuklarına gönderilir. Buradaki fikir güncellemeler sadece gerekli olduğu zaman aşağıya göndermektir ki bu da operasyonların verimli olduğunu garantiler.

Aşağıdaki resim $\text{sum}_a(a, b)$ değerini hesaplarırken ağacın nasıl değiştiğini gösterecektir. Dikdörtgen değeri değişen düğümleri gösterir çünkü bir tembel güncelleme aşağıya doğru ilerler.



Bazen tembel güncellemeleri birleştirmenin gerektiğine dikkat edelim. Bu zaten tembel güncellemeye sahip bir düğüme başka bir tembel güncelleme verildiği zaman olur. Toplamları hesaplarırken tembel güncellemeleri birleştirmek kolaydır çünkü z_1 and z_2 güncellemeleri zaten $z_1 + z_2$ güncellemesine denk gelir.

Polinom Güncellemeleri

Tembel güncellemeleri genelleterek aralıkları polinom kullanarak

$$p(u) = t_k u^k + t_{k-1} u^{k-1} + \dots + t_0$$

formunda güncelleyebiliriz.

Bu durumda $[a, b]$ aralığındaki i pozisyonunda olan değeri güncellemek $p(i - a)$ olur. Örneğin $p(u) = u + 1$ polinomunu $[a, b]$ aralığına eklemek a pozisyonundaki değeri 1 arttırmak, $a + 1$ pozisyonundaki değeri 2 arttırmak şeklinde ilerler.

Polinom güncellemelerini desteklemek için her düğümün $k + 2$ değeri olur ve k polinomun derecesine eşittir. s değeri ise aralıktaki elemanların toplamına denk gelir ve z_0, z_1, \dots, z_k değerleri ise de bir tembel güncellemeye denk gelen bir polinomun katsayılarını gösterir.

Şimdi, bir $[x, y]$ aralığındaki değerlerin toplamı

$$s + \sum_{u=0}^{y-x} z_k u^k + z_{k-1} u^{k-1} + \dots + z_0.$$

Böyle bir toplamın değeri verimli bir şekilde toplam formülleri kullanarak hesaplanabilir. Örneğin z_0 terimi $(y-x+1)z_0$ toplamına karşılık gelir ve z_1u ise de

$$z_1(0+1+\dots+y-x) = z_1 \frac{(y-x)(y-x+1)}{2}$$

toplamına karşılık gelir.

Ağaçta bir güncellemeyi yayarken $p(u)$ indisleri değişir çünkü her $[x, y]$ aralığında, değerler $u = 0, 1, \dots, y-x$ için hesaplanır. Fakat bu bir problem değil çünkü $p'(u) = p(u+h)$ $p(u)$ polinomuna eşit dereceye sahip bir polinomdur. Örneğin eğer $p(u) = t_2u^2 + t_1u - t_0$ ise

$$p'(u) = t_2(u+h)^2 + t_1(u+h) - t_0 = t_2u^2 + (2ht_2 + t_1)u + t_2h^2 + t_1h - t_0.$$

28.2 Dinamik Ağaçlar

Sıradan bir segment ağacı statiktir yani her düğümün dizide sabit bir pozisyonu vardır ve ağaç sabit miktarda hafızaya ihtiyaç duyar. **Dinamik segment ağacında** hafıza sadece algoritmada kullanılan düğümlere verilir ki bu da büyük miktarda hafıza tasarrufu sağlar.

Dinamik ağaçtaki düğümler yapı olarak gösterilebilir:

```
struct node {
    int value;
    int x, y;
    node *left, *right;
    node(int v, int x, int y) : value(v), x(x), y(y) {}
};
```

Burada value düğümün değeri, $[x, y]$ düğümde bulunan aralık ve left ve right noktaları da sol ve sağ alt ağaçlarını gösterir.

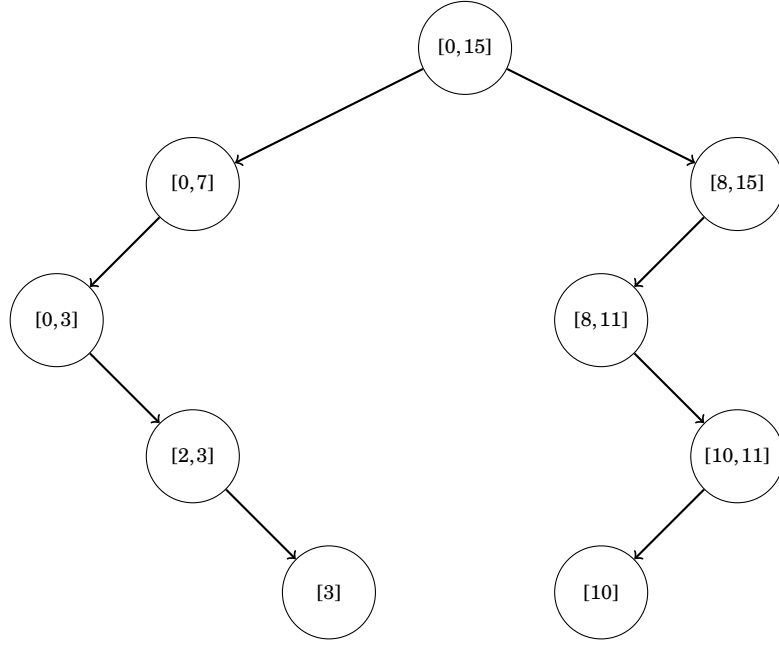
Bundan sonra düğümler şekildeki gibi oluşturulabilir:

```
// yeni düğüm şoludur
node *x = new node(0, 0, 15);
// gdeeri gşdeitir
x->value = 5;
```

Seyrek Segment Ağaçları (Sparse Segment Trees)

Dinamik bir segment ağacı eğer dizisi *seyrekse* işe yarar yani izin verilen $[0, n-1]$ indislerinin aralığının büyük olması çünkü çoğu dizi değeri 0'dan oluşur. Sıradan bir segment ağacı $O(n)$ hafıza kullanırken dinamik segment ağacı sadece $O(k \log n)$ hafıza kullanır ki k toplam uygulanan operasyon sayısıdır.

Bir **seyrek segment ağacı** başta sadece değeri 0 olan $[0, n-1]$ 1 düğüm içerir yani her düğümün değeri 0'dır. Güncellemelerden sonra yeni düğümler dinamik olarak ağaca eklenir. Örneğin $n = 16$ ise 3 ve 10. pozisyonundaki elemanlar değiştirilir ve ağaç aşağıdaki düğümleri içerir:



Kök düğümden herhangi bir yaprağa olan herhangi bir yol $O(\log n)$ düğüm içerir yani her operasyonda en fazla $O(\log n)$ düğüm eklenir. Böylece k operasyon sonra ağaç en fazla $O(k \log n)$ düğüm içerir.

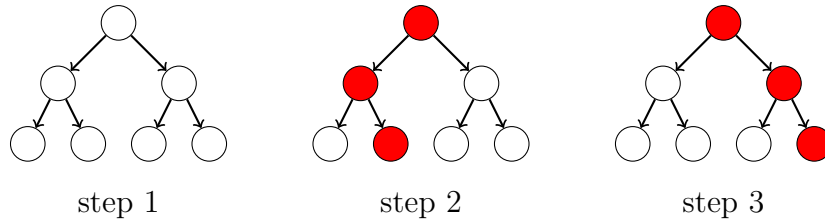
Algoritmanın başında bütün elemanların güncellenmesini gerektiğini biliyorsak dinamik segment ağacı gerekli olmayabilir çünkü indis sıkıştırmasıyla sıradan bir segment ağacı kullanabiliriz (Bölüm 9.4). Fakat indislerin algoritma sırasında oluşturulduğu durumda bu olası değildir.

Sürekli Segment Ağaçları (Persistent Segment Trees)

Dinamik implementasyon kullanarak ağacın *değişiklik geçmişi*ni tutacak bir **sürekli segment ağacı** oluşturabiliriz. Bu tip bir implementasyonda verimli bir şekilde ağacın algoritma sırasında olduğu her duruma erişebiliriz.

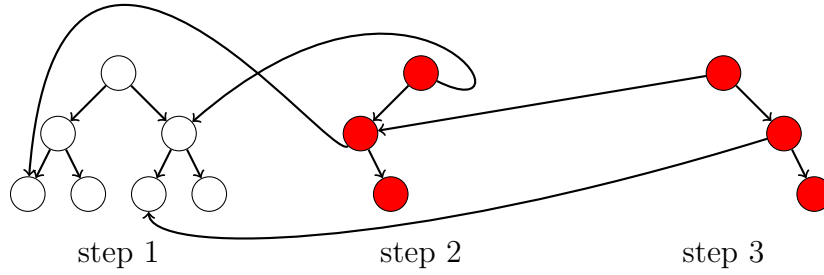
Değişiklik geçmişi olduğu zaman önceki ağaçlardan herhangi birinde klasik bir segment ağacı gibi sorgu yapabiliriz çünkü ağacın bütün yapısı tutuluyor. Aynı zamanda önceki ağaçlardan yeni ağaçlar oluşturup onları ayrı bir şekilde değiştirebiliriz.

Aşağıdaki güncelleme dizisine bakalım. Burada kırmızı düğümler değişeceğini ve diğer düğümler aynı kalacağını söyler:



Her güncellemeden sonra ağaçtaki çoğu düğüm aynı kalar yani değişiklik geçmişi tutmanın bir yolu her ağacı geçmişte yeni düğümlerin kombinasyonu şeklinde

ve önceki ağaçların alt ağaçları şeklinde tutabiliriz. Örneğin aşağıdaki durumda değişiklik geçmişi şekildeki gibi tutulur:



Her önceki ağacın yapısı kök düğümde başlayan işaretçilerle yeniden oluşturulabilir. Her operasyon sadece ağaca $O(\log n)$ yeni düğüm eklediği için ağacın bütün değişiklik geçmişinin tutabiliriz.

28.3 Veri Yapıları

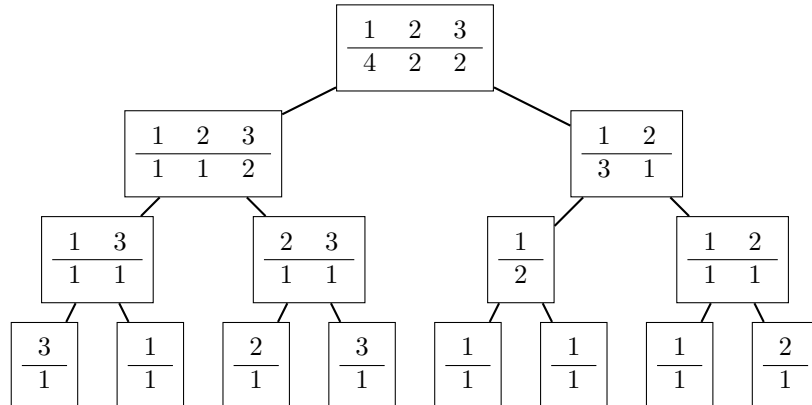
Bir segment ağacındaki düğümler tek değer yerine aynı zamanda *aralık* hakkında bilgi tutan *veri yapıları* da tutabilir. Böyle bir ağaçta operasyonlar $O(f(n)\log n)$ zaman alır ki burada $f(n)$ operasyon sırasında bir düğümü işlemek için gereken zamandır.

Örneğin " x elemanı $[a, b]$ aralığında kaç defa bulunur?" tipindeki sorguları destekleyen bir segment ağacı düşünelim. Örneğin 1 elemanı aşağıdaki aralıkta 3 defa bulunur:

3	1	2	3	1	1	1	2
---	---	---	---	---	---	---	---

Bu tip sorguları desteklemek için her düğümün bir veri yapısına sahip olduğu bir segment ağacı oluştururuz. Bu düğümlerdeki veri yapısına onların bulunduğu aralıkta x elemanının kaç defa bulunduğunu sorabiliriz. Bu ağacı kullanarak sorunun cevabı aralıkta bulunan düğümlerden gelen cevaplar birleştirilerek bulunabilir.

Örneğin aşağıdaki segment ağacı yukarıdaki diziye karşılık gelir:



Her düğüm bir `map` yapısı içerecek şekilde bir ağaç oluşturabiliriz. Bu durumda her düğümü işlemek için gereken zaman $O(\log n)$ yani bir sorgunun toplam zaman karmaşıklığı $O(\log^2 n)$ olur. Ağaç $O(n \log n)$ hafıza kullanır çünkü $O(\log n)$ level vardır ve her level $O(n)$ eleman içerir.

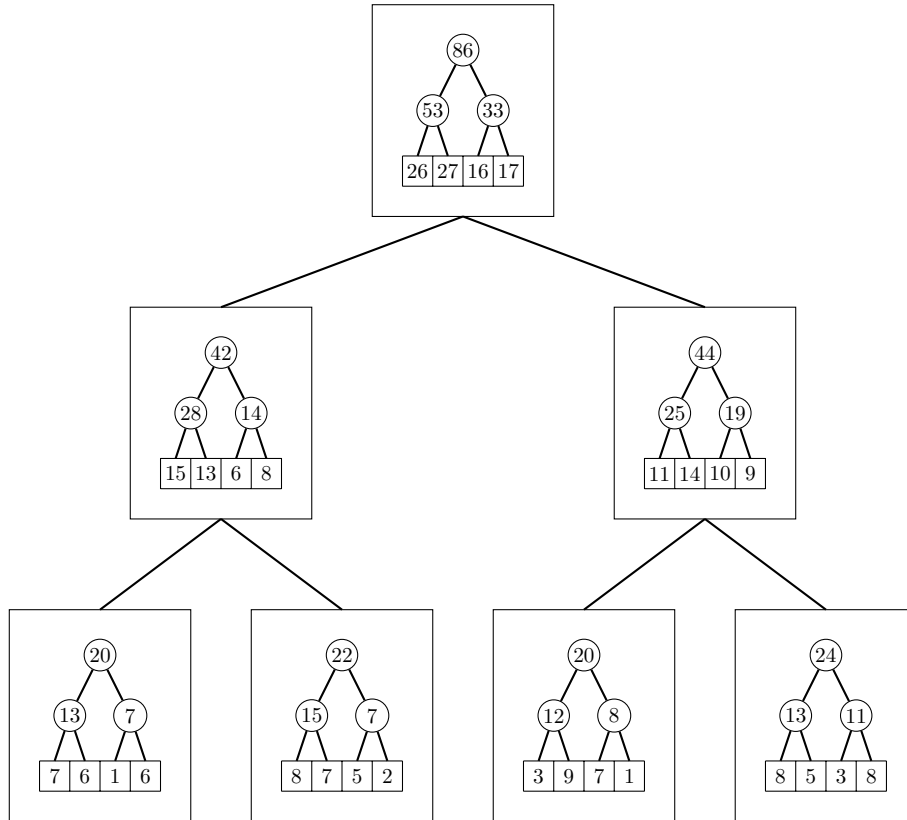
28.4 İki-Boyutluluk

Bir **iki-boyutlu segment ağacı** iki boyutlu dizilerin dikdörtgen alt dizilerini destekleyen sorguları sağlar. Bu tip bir ağaç iç içe segment ağaçlarıyla yapılabilir: dizinin satırlarına karşılık büyük bir ağaç ve küçük ağaçtaki her düğüm bir sütuna karşılık gelir.

Örneğin

7	6	1	6
8	7	5	2
3	9	7	1
8	5	3	8

dizisindeki herhangi bir alt dizinin toplamı aşağıdaki segment ağacıyla hesaplanabilir:



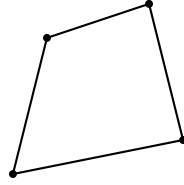
İki-boyutlu segment ağacının operasyonları $O(\log^2 n)$ zaman alır çünkü büyük ağaç ve her küçük ağaç $O(\log n)$ level içerir. Ağaç $O(n^2)$ hafıza içerir çünkü her küçük ağaç $O(n)$ tane değer içerir.

Bölüm 29

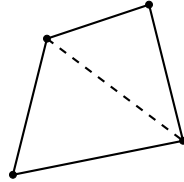
Geometri

Geometri sorularında kolay bir şekilde koda dökülebilecek bir çözüm bulmak genelde zordur ve özel durum sayısı azdır.

Örneğin, bize bir dörtgen (dört köşeden oluşan şekil) verildiğini ve amacımızın alanını hesaplamak olan bir problem verildiğini düşünelim. Örneğin bu problem için olası bir girdi aşağıdaki gibidir:



Probleme bakmanın bir yolu, dörtgeni karşılıklı iki köşesinden bir çizgi çekerek iki üçgene bölmektir:

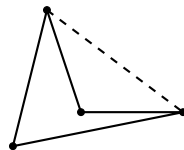


Bundan sonra üçgenin alanlarını hesaplamak yeterlidir. Üçgenin alanı hesaplanabilir. Örneğin **Heron Formülünü** kullanarak a , b ve c 'nin üçgenin kenarlarının uzunluğu ve $s = (a + b + c)/2$ olduğu kabul ederek

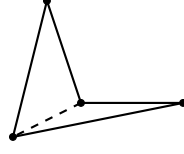
$$\sqrt{s(s-a)(s-b)(s-c)}$$

şeklinde hesaplanabilir.

Bu problemi çözmenin olası bir yolu var fakat bir sorunu da var. Dörtgeni nasıl üçgenlere ayıracağız? Bazı durumlarda öylesine karşılıklı iki köşe alıp ayıramayız. Örneğin aşağıdaki durumda bölen çizgi dörtgenin *dışarısında* kalıyor:



Fakat, çizgiyi çizmenin başka bir yolu daha var. However, another way to draw the line works:



Bu çizgilerden hangisinin doğru olduğunu görmemiz biz insanlar için kolay olsa da bilgisayar için zor oluyor.

Fakat, bu problemi bir programcıya rahat olacak şekilde başka bir yolla çözebiliriz. Köşeleri (x_1, y_1) , (x_2, y_2) , (x_3, y_3) ve (x_4, y_4) olan bir dörtgenin alanını

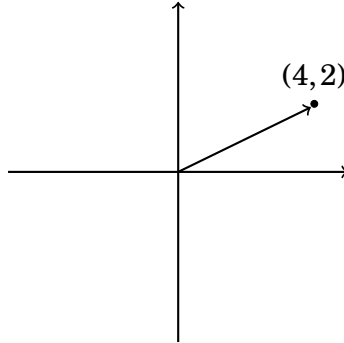
$$x_1y_2 - x_2y_1 + x_2y_3 - x_3y_2 + x_3y_4 - x_4y_3 + x_4y_1 - x_1y_4$$

formülü ile hesaplayabiliriz. Bu formül yapması kolaydır çünkü hiç özel durum yoktur ve bu formül *all* çokgenlere uyarlıdır.

29.1 Karmaşık Sayılar

Bir **karmaşık sayı** $x + yi$ formunda olan bir sayıdır. Burada $i = \sqrt{-1}$ olup **sanal birimdir**. Karmaşık sayının geometrik gösterimi ya iki boyutlu bir (x, y) noktadır veya orijinden bir noktaya (x, y) olan bir vektördür.

Örneğin $4 + 2i$ aşağıdaki nokta ve vektöre karşılık gelir.



C++ karmaşık sayı sınıfı **complex** geometri sorularını çözerken işe yarar. Bu sınıfı kullanarak noktaları ve vektörleri karmaşık sayı şeklinde gösterebiliriz ve sınıf geometride işe yarayacak araçları bulundurur.

Aşağıdaki kodda, C bir koordinatın tipidir ve P ise bir noktanın veya vektörün tipidir. Bunla beraber kod X ve Y makrolarını x ve y koordinatlarına karşılık gelmesi için tanımlar.

```
typedef long long C;  
typedef complex<C> P;  
#define X real()  
#define Y imag()
```

Örneğin aşağıdaki kod bir $p = (4, 2)$ noktası tanımlar ve x ve y koordinatlarını yazdırır.

```
P p = {4,2};  
cout << p.X << " " << p.Y << "\n"; // 4 2
```

Aşağıdaki kod $v = (3, 1)$ ve $u = (2, 2)$ vektörlerini tanımlar ve bundan sonra $s = v + u$ toplamını hesaplar.

```
P v = {3,1};  
P u = {2,2};  
P s = v+u;  
cout << s.X << " " << s.Y << "\n"; // 5 3
```

Pratikte genelde uygun koordinat tipi ya `long long` (tam sayı) ya da `long double` (reel sayı) olur. Tam sayı kullanabildiği zaman tam sayı kullanmak iyi bir fikirdir çünkü tam sayılarla hesaplamalar kesindir. Eğer reel sayılar gerekirse sayıları karşılaştırırken kesinlik (precision) hatası vardır. a ve b sayılarının eşit olup olmadığını kontrol etmenin bir yolu onları $|a - b| < \epsilon$ ile kontrol etmektir. Burada ϵ küçük bir sayıdır (Örneğin $\epsilon = 10^{-9}$).

Fonksiyonlar

Aşağıdaki örneklerde koordinat tipi `long double` idir/

`abs(v)` fonksiyonu bir $v = (x, y)$ vektörünün uzunluğunu $|v| \sqrt{x^2 + y^2}$ formülünü kullanarak hesaplar. Fonksiyon aynı zamanda (x_1, y_1) ve (x_2, y_2) noktaları arasında mesafeyi hesaplamak için de kullanılabilir çünkü bu mesafe $(x_2 - x_1, y_2 - y_1)$ vektörünün uzunluğuna eşittir.

Aşağıdaki kod $(4, 2)$ ve $(3, -1)$ noktaları arasındaki mesafeleri hesaplar:

```
P a = {4,2};  
P b = {3,-1};  
cout << abs(b-a) << "\n"; // 3.16228
```

`arg(v)` fonksiyonu $v = (x, y)$ vektörünün x eksenine olan açığı hesaplar. Fonksiyon açığı radyan cinsinden olur. r radyan, $180r/\pi$ dereceye eşittir. Sağ gösteren bir vektörün açısı 0'dır ve açılar saat yönünde azalır ve saat yönünün tersinde artar.

`polar(s, a)` fonksiyonu uzunluğu s olup a açısında olan bir vektör oluşturur. Vektör, 1 uzunluğunda a açısına sahip bir vektörler çarpılarak a açısı kadar döndürülebilir.

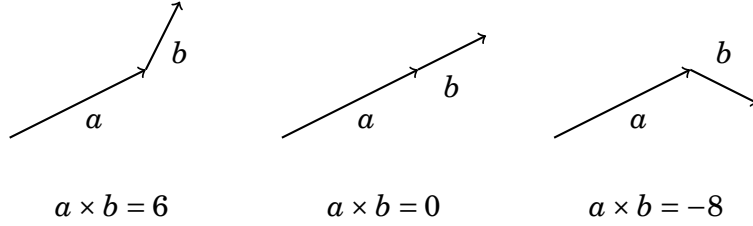
Aşağıdaki kod $(4, 2)$ vektörünün açısını hesaplar, $1/2$ radyan miktarında saat yönünün tersine döndürür ve sonra açığı yeniden hesaplar:

```
P v = {4,2};  
cout << arg(v) << "\n"; // 0.463648  
v *= polar(1.0, 0.5);  
cout << arg(v) << "\n"; // 0.963648
```

29.2 Noktalar ve Çizgiler

Vektörel çarpım (cross product) yani $a \times b$, $a = (x_1, y_1)$ ve $b = (x_2, y_2)$ vektörlerinin $x_1y_2 - x_2y_1$ formülünü kullanarak çarpılmasıdır. Vektörel çarpım b 'nin a 'dan hemen sonra yerleştirilmesinden sonra b 'nin ya sola döndüğünü (pozitif değer), ya dönmediğini (sıfır) ya da sağa döndüğünü (negatif değer) söyler.

Aşağıdaki resim yukarıdaki durumları gösterir:



Örneğin ilk durumda $a = (4, 2)$ ve $b = (1, 2)$ idir. Aşağıdaki kod complex sınıfını kullanarak vektör çarpımını hesaplar:

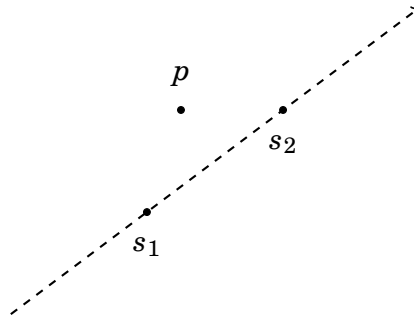
```
P a = {4,2};  
P b = {1,2};  
C p = (conj(a)*b).Y; // 6
```

Yukarıdaki kod çalışır çünkü conj fonksiyonu bir vektörün y koordinatını yok sayar ve $(x_1, -y_1)$ ve (x_2, y_2) vektörleri birbirleriyle çarpıldığı zaman sonucun y koordinatı $x_1y_2 - x_2y_1$ olur.

Nokta Lokasyonu

Vektörel çarpımlar bir noktanın bir çizginin solunda mı yoksa sağında mı olduğunu kontrol etmek için de kullanılabilir. Diyelim ki çizgi s_1 ve s_2 noktalarından gidiyor ve biz s_1 'ten s_2 'ye bakıp noktanın p olduğunu biliyoruz.

Örneğin aşağıdaki resimde p çizginin sol tarafındadır:

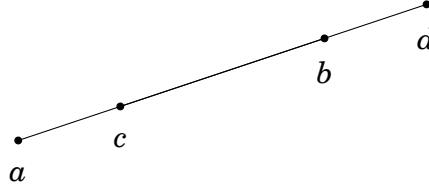


$(p - s_1) \times (p - s_2)$ vektörel çarpımı bize p noktasının yerini söyler. Eğer vektörel çarpım pozitif ise p sol tarafta bulunur ve eğer vektörel çarpım negatif ise p sağ tarafta bulunur. En sonda eğer vektörel çarpım 0 ise s_1 , s_2 ve p noktaları aynı çizgi üzerindedir.

Çizgi Kesişimi (Line Segment Intersection)

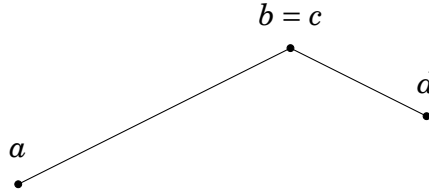
Şimdi ab ve cd isimli iki çizginin kesişip kesişmediğine bakalım. Olası durumlar:

1. *Durum:* Çizgiler aynı çizgi üzerindeler ve böylece üst üste geliyorlar. Bu durumda sonsuz tane kesişim noktası vardır. Örneğin aşağıdaki resimde c ve b noktaları arasındaki bütün noktalar kesişim noktalarıdır:



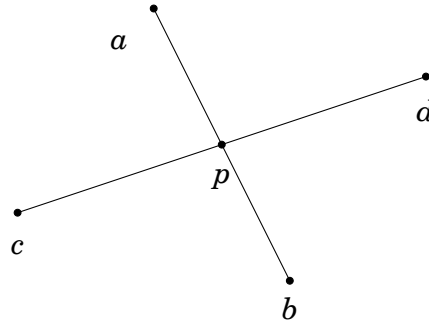
Bu durumda vektörel çarpımı kullanarak bütün noktaların aynı çizgi üzerinde olup olmadığını kontrol edebiliriz. Bundan sonra noktaları sıralarız ve çizgilerin birbirleriyle üst üste gelip gelmediğini kontrol ederiz.

2. *Durum:* Çizgilerin tek bir kesişim noktası olan bir ortak köşesi vardır. Örneğin, aşağıdaki resimde kesişim noktası $b = c$:



Bu durumda kontrol etmek kolaydır çünkü sadece kesişim noktası için 4 olası yer vardır: $a = c$, $a = d$, $b = c$ ve $b = d$.

3. *Durum:* Herhangi bir çizginin köşesi olmayan tam bir tane kesişim noktası vardır. Aşağıdaki resimde p noktası kesişim noktasıdır:



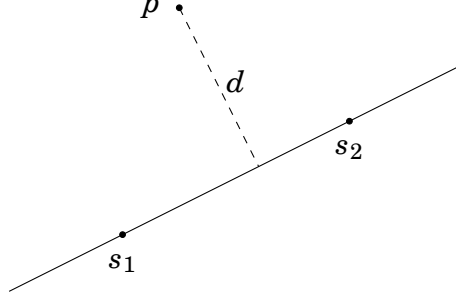
Bu durumda çizgiler, c ve d noktalarının ikisi de a ve b 'den geçen bir çizginin farklı yönlerindeyse kesişirler. Bunu kontrol etmek için vektörel çarpım kullanabiliriz.

Bir Çizgiden Nokta Mesafesi

Vektörel çarpımlarının başka bir özelliği, üçgenin alanının

$$\frac{|(a - c) \times (b - c)|}{2}$$

formülü ile hesaplanabiliyor olmasıdır ki burada a , b ve c üçgenin köşeleridir. Bu durumu kullanarak bir nokta ve çizgi arasındaki en kısa mesafeyi hesaplayabiliriz. Örneğin aşağıdaki resimde p noktası ile s_1 ve s_2 noktaları arasındaki çizginin en kısa mesafe d 'dir:

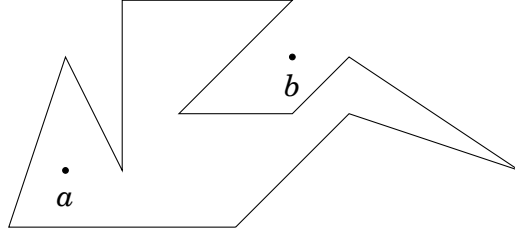


Köşeleri s_1 , s_2 and p olan üçgenin alanı iki farklı şekilde hesaplanabilir: Bunlar $\frac{1}{2}|s_2 - s_1|d$ ve $\frac{1}{2}((s_1 - p) \times (s_2 - p))$ olur. Böylece en kısa mesafe

$$d = \frac{(s_1 - p) \times (s_2 - p)}{|s_2 - s_1|}.$$

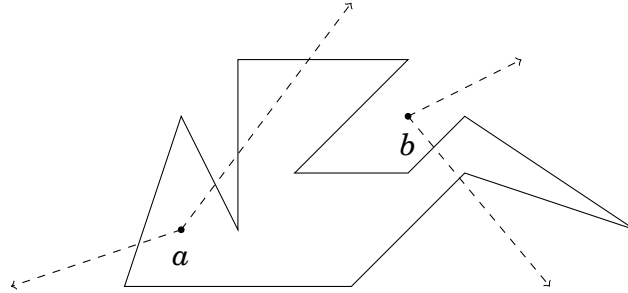
Çokgen İçinde Nokta

Şimdi bir noktanın bir çokgenin dışında mı içinde mi olduğuna bakalım. Örneğin aşağıdaki resimde a noktası çokgenin içindeyken b noktası çokgenin dışındadır.



Problemi çözmenin rahat bir yolu noktadan herhangi bir yöne doğru bir *ışın* gönderip çokgenin sınırlarına kaç defa dokunduğuna bakmaktır. Eğer sayı tekse nokta çokgenin içinde ve eğer sayı çiftse nokta çokgenin dışındadır.

Örneğin aşağıdaki ışınları gönderebiliriz:



a 'dan çıkan ışınlar 1 ve 3 defa çokgenin sınırlarına dokunur yani a çokgenin içindedir. Bunlar beraber b 'den çıkan ışınlar çokgenin sınırlarına 0 ve 2 defa dokunur yani b çokgenin dışındadır.

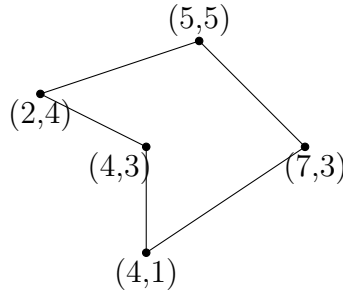
29.3 Çokgen Alanı

Bir çokgenin formülünü hesaplamamanın genel bir yolu (bazen **ayakkabı bağı formülü**(shoelace formula) olarak söylenir) şöyle işler: köşelerin $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$, ..., $p_n = (x_n, y_n)$ olduğu durumda ve p_i ve p_{i+1} köşelerinin çokgenin sınırlarındaki ardışık köşeler olacağı sırasıyla (ilk ve son köşe aynıdır yani $p_1 = p_n$)

$$\frac{1}{2} \left| \sum_{i=1}^{n-1} (p_i \times p_{i+1}) \right| = \frac{1}{2} \left| \sum_{i=1}^{n-1} (x_i y_{i+1} - x_{i+1} y_i) \right|$$

formüle ile hesaplanır.

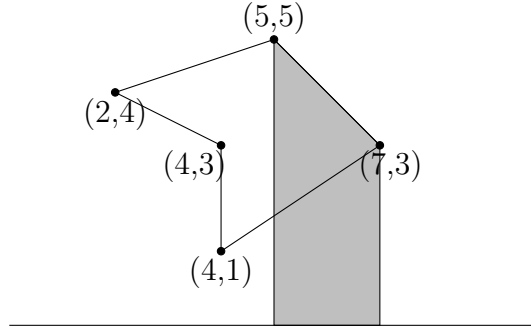
Örneğin aşağıdaki çokgenin alanı:



is

$$\frac{|(2 \cdot 5 - 5 \cdot 4) + (5 \cdot 3 - 7 \cdot 5) + (7 \cdot 1 - 4 \cdot 3) + (4 \cdot 3 - 4 \cdot 1) + (4 \cdot 4 - 2 \cdot 3)|}{2} = 17/2.$$

Formülün fikri bir tane kenarı çokgenin bir kenarı olan ve diğer kenarı da $y = 0$ yatay çizgisinde olan bütün yamuklardan geçmektir:



Böyle bir yamuğun alanı, çokgenin köşelerinin p_i ve p_{i+1} olduğu durumda

$$(x_{i+1} - x_i) \frac{y_i + y_{i+1}}{2}$$

olur. Eğer $x_{i+1} > x_i$ ise alan pozitifdir ve eğer $x_{i+1} < x_i$ ise alan negatiftir.

Bir çokgenin alanı bu tip yamukların alanının toplamıdır ki bu da

$$\left| \sum_{i=1}^{n-1} (x_{i+1} - x_i) \frac{y_i + y_{i+1}}{2} \right| = \frac{1}{2} \left| \sum_{i=1}^{n-1} (x_i y_{i+1} - x_{i+1} y_i) \right|$$

formülünü verir.

Bu arada toplamın mutlak değeri alınmasına dikkat edilmesi lazım çünkü toplamın değeri çokgenin sınırları üzerinde saat yönünde veya saat yönünün tersinde yürümemize göre pozitif veya negatif olabilir.

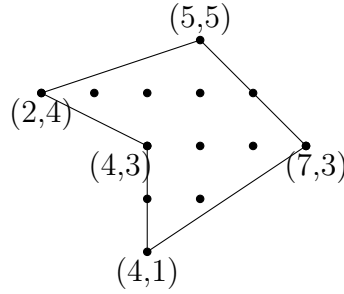
Pick'in Teoremi

Pick'in teoremi bütün köşeleri tamsayı koordinatlarda olan bir çokgenin alanını bulmamıza yarayan başka bir yol verir. Pick'in Teoremi'ne göre çokgenin alanı

$$a + b/2 - 1$$

olur ki burada a çokgenin içindeki tamsayı noktalarını ve b de çokgenin sınırları üzerindeki tamsayı noktalarını verir.

Örneğin, aşağıdaki çokgenin alanı:



$6 + 7/2 - 1 = 17/2$ olur.

29.4 Uzaklık Fonksiyonları

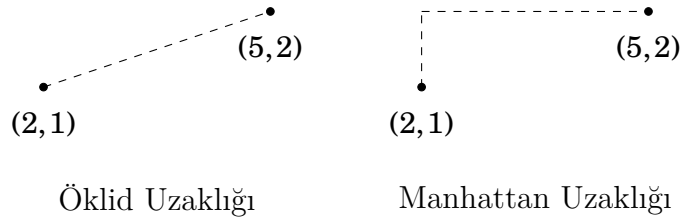
Bir **uzaklık fonksiyonu** iki nokta arasındaki mesafeyi tanımlar. Genelde kullanılan uzaklık fonksiyonu **Öklid uzaklığı** yani (x_1, y_1) ve (x_2, y_2) noktaları arasındaki uzaklığının

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

olduğu fonksiyondur. Başka bir uzaklık fonksiyonu ise **Manhattan uzaklığı**dır ve burada (x_1, y_1) ve (x_2, y_2) noktaları arasındaki uzaklık

$$|x_1 - x_2| + |y_1 - y_2|.$$

olur. Örneğin aşağıdaki resimde:



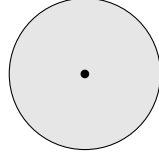
Noktalar arasındaki Öklid uzaklığı

$$\sqrt{(5-2)^2 + (2-1)^2} = \sqrt{10}$$

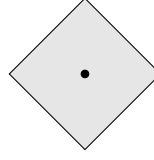
ve Manhattan uzaklığı

$$|5 - 2| + |2 - 1| = 4.$$

olur. Aşağıdaki resimde merkez noktadan 1 birim uzaklığındaki bölgeler, Öklid ve Manhattan uzaklıkları kullanılarak gösterilmiştir:



Öklid Uzaklığı

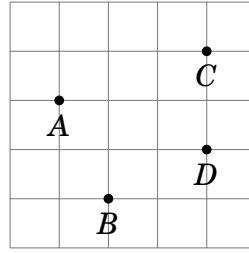


Manhattan Uzaklığı

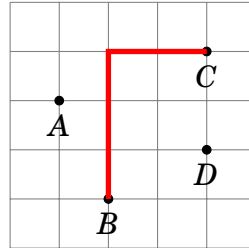
Koordinatları Döndürmek

Baze problemler, Öklid uzaklığı kullanmak yerine Manhattan uzaklığı kullanılarak çözmek daha kolaydır. Örneğin iki boyutlu bir düzlemde bize n tane nokta verildiğini ve amacımızın herhangi iki nokta arasındaki en fazla Manhattan uzaklığını bulmamızın gerektiği bir soru olsun.

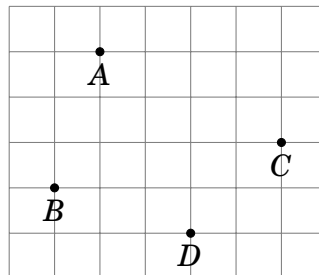
Örneğin aşağıdaki nokta kümesine bakalım:



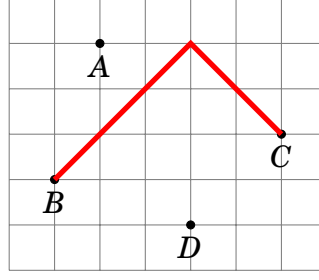
Maksimum Manhattan uzaklığı 5 olup B ve C arasındadır:



Manhattan uzaklığıyla alakalı işe yarar bir teknik bütün koordinatları 45 derece döndürmektir. Böylece bir (x, y) noktası $(x + y, y - x)$ noktasına dönüşür. Örneğin yukarıdaki noktaları döndürdükten sonra sonuç:



Maksimum mesafe aşağıdaki gibi olur:



Diyelim ki elimizde $p_1 = (x_1, y_1)$ ve $p_2 = (x_2, y_2)$ noktaları var ve bunların döndürülmüş koordinatları $p'_1 = (x'_1, y'_1)$ ve $p'_2 = (x'_2, y'_2)$ olsun. Şimdi p_1 ve p_2 arasındaki Manhattan uzaklığını göstermenin iki yolu vardır:

$$|x_1 - x_2| + |y_1 - y_2| = \max(|x'_1 - x'_2|, |y'_1 - y'_2|)$$

Örneğin $p_1 = (1, 0)$ ve $p_2 = (3, 3)$ ise döndürülmüş koordinatlar $p'_1 = (1, -1)$ ve $p'_2 = (6, 0)$ olur ve Manhattan mesafesi

$$|1 - 3| + |0 - 3| = \max(|1 - 6|, |-1 - 0|) = 5$$

olur.

Döndürülmüş koordinatlar Manhattan uzaklığıyla çalışmak için basit bir yol sağlar çünkü x ve y koordinatlarını ayrı ayrı düşünebiliriz. İki nokta arasındaki Manhattan uzaklığını maksimum yapmak için döndürülmüş koordinatları

$$\max(|x'_1 - x'_2|, |y'_1 - y'_2|)$$

değerini maksimum yapan iki tane nokta bulmamız gerekir. Bu kolaydır çünkü döndürülmüş koordinatların ya yatay farkı ya da dikey farkından biri maksimum olmak zorundadır.

Bölüm 30

Çizgi Süpürme Algoritmaları (Sweep Line Algorithms)

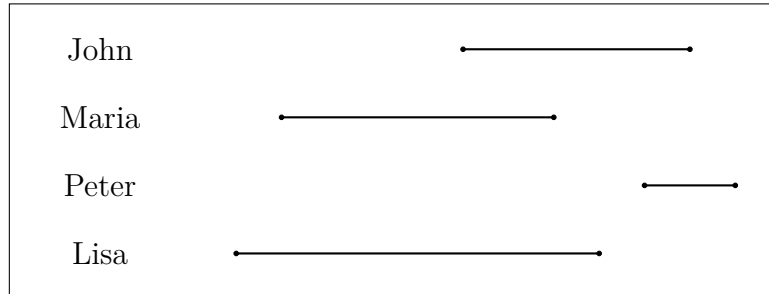
Çoğu geometrik problem **çizgi süpürme** algoritmaları ile çözülebilir. Bu tip algoritmalarındaki fikir, prbolemi düzlemde noktalara karşılık gelen bir durumlar kümesinde göstermektir. Durumlar x veya y koordinatlarına göre artan sırada işlenirler.

Örneğin, şu problemi düşünelim: n tane çalışan olan bir şirket var ve her çalışanın belirli bir günde giriş ve çıkış saatlerini biliyoruz. Amacımız ofiste herhangi bir zamanda bulunan en fazla çalışan sayısını bulmak olsun.

Problem durumu modelleyerek çözülebilir. Böylece her çalışan giriş ve çıkış saatlerine göre iki durumu olur. Durumları sıraladıktan sonra onlardan geçip ofisteki insan sayısını tutabiliriz. Örneğin

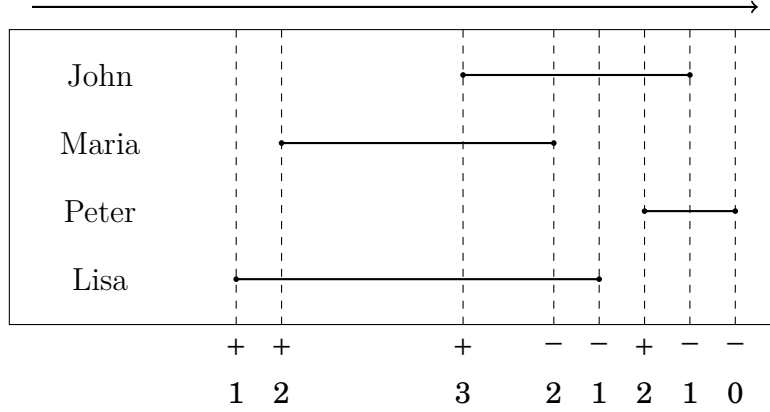
kişi	giriş zamanı	çıkış zamanı
John	10	15
Maria	6	12
Peter	14	16
Lisa	5	13

tablosu aşağıdaki durumlara karşılık gelir:



Soldan sağa doğru durumlardan geçeriz ve aynı zamanda bir sayaç tutarız. Ne zaman 1 insan gelirse sayacı bir artırırız ve ne zaman birisi ayrılırsa sayacı 1 azaltırız. Sorunun çözümü algoritma sırasında sayacın maksimum değeridir.

Örneğin durumlar şekildeki gibi işlenir:

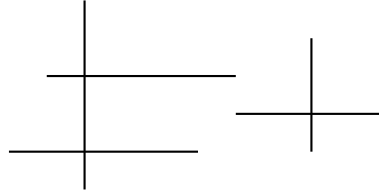


+ ve - sembolleri sayacın arttırılıp azaltılacağını belirtir ve sayacın değeri gösterilir. Sayacın maksimum değeri 3 olup John'un gelişi ile Maria'nın çıkışı arasındaki zamandır.

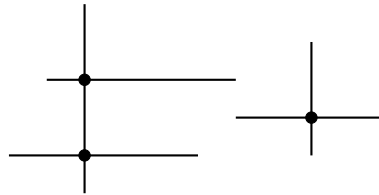
Algoritmanın çalışma zamanı $O(n \log n)$ olur çünkü durumları sıralamak $O(n \log n)$ ve algoritmanın geri kalan kısmı $O(n)$ zaman alır.

30.1 Kesişim Noktaları

n tane çizginin bulunduğu bir kümede çizgilerin toplam kesişim noktalarını saymamız gerektiğini düşünelim. Burada her çizgi ya yatay ya da dikeydir. Örneğin çizgiler



üç tane kesişim noktası vardır:

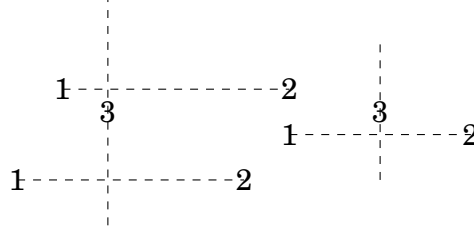


Soruyu $O(n^2)$ zamanda çözmek kolaydır çünkü bütün olası çizgi çiftlerini inceleyip kesişip kesişmediklerini inceleyebiliriz. Fakat bu problemi daha verimli bir şekilde $O(n \log n)$ zamanda çizgi süpürme algoritması ve aralık sorgu veri yapısıyla yapabiliriz.

Buradaki fikir, soldan sağa doğru çizgilerin uç noktalarını işleyip üç durumda odaklanmaktır:

- (1) yatay çizgi başlıyor
- (2) yatay çizgi bitiyor
- (3) dikey çizgi

Aşağıdaki durumlar örneğe karşılık gelir:



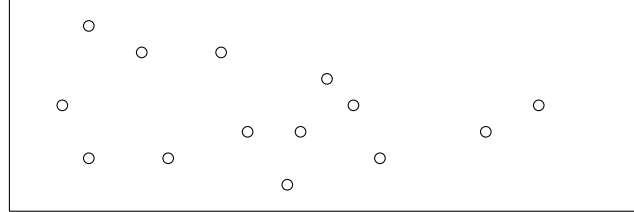
Burada, durumlardan soldan sağa gidip aktif yatay çizginin olduğu yerlerdeki y koordinatlarını tutan bir kümeyi tutan bir veri yapısı kullanırız. 1. durumda çizginin y koordinatını kümeye ekleriz ve 2. durumda y koordinatını kümeden çıkarırız.

Kesişim noktaları 3. durumda hesaplanır. y_1 ve y_2 noktaları arasında dikey bir çizgi varsa y koordinatı y_1 ile y_2 arasında olan aktif yatay çizgi sayısını sayarız ve bu numarayı toplam kesişim sayısına ekleriz.

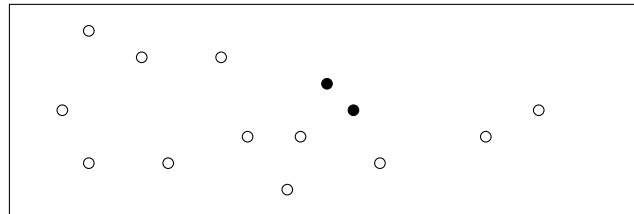
Yatay çizgilerin y koordinatlarını hesaplamak için indis sıkıştırmasıyla ikili indeksli ağaç veya segment ağacı kullanabiliriz. Bu yapılar kullanıldığı zaman her durumu işlemek $O(\log n)$ zaman alır ve yani algoritmanın toplam zamanı $O(n \log n)$ olur.

30.2 En Yakın Çift Problemi

n tane noktadan oluşan bir kümede sonraki amacımız Öklid uzaklığı en kısa olan iki noktayı bulmaktır. Örneğin noktalar



o zaman şu noktaları bulmamız gerekir:



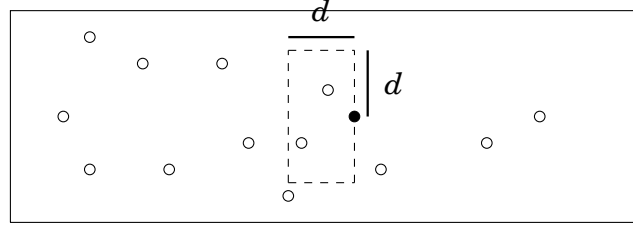
Bu da çizgi süpürme algoritması¹ ile çözülebilen başka bir sorudur. Burada soldan sağa doğru noktalardan geçeriz ve şu ana kadar geçtiğimiz noktalar arasında en kısa mesafeyi gösteren d mesafesini tutarız. Her noktada s en yakın

¹Bu yöntem haricinde $O(n \log n)$ zamanda çalışan bir böl ve yönet algoritması da vardır [56]. Bu algoritma noktaları iki kümeye ayırır ve her küme için soruyu çözer.

noktayı buluruz. Eğer mesafe d 'den az ise yeni minimum mesafeyi bulmuş oluruz ve d değerini güncelleriz.

Eğer şu anki nokta (x, y) ise ve solda mesafesi d 'den daha az olan bir nokta varsa bu tip bir noktanın x koordinatı $[x-d, x]$ ve y koordinatı $[y-d, y+d]$ olacaktır. Böylece sadece bu aralıktaki noktaları kontrol etmemiz yeterli olacağından algoritma verimli olur.

Örneğin aşağıdaki resimde çizgili çizgilerle belirlenen alan, aktif noktadan d uzaklığı içinde bulunan noktaları içerir:



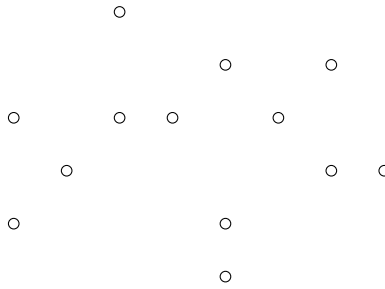
Bu algoritmanın verimliliği bölgenin her zaman $O(1)$ nokta içermesinden gelir. Bu noktaları $O(\log n)$ zamanda x koordinatı $[x-d, x]$ arasında olan noktaları küme halinde y koordinatlarına göre artan halde tutarak yapmak mümkündür.

Bu algoritmanın zaman karmaşıklığı $O(n \log n)$ olur çünkü n tane noktadan geçeriz ve her noktada sola en yakın noktayı $O(\log n)$ zamanda buluruz.

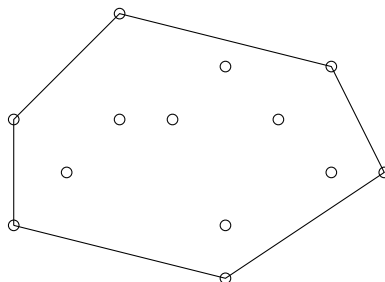
30.3 Convex hull Problemi

Bir **convex hull** bir kümedeki bütün noktaları içeren en küçük konveks çokgendir. Konvekslik, çokgendeki herhangi iki nokta arasındaki bir çizginin tamamıyla çokgenin içinde olduğunu söyler.

Örneğin



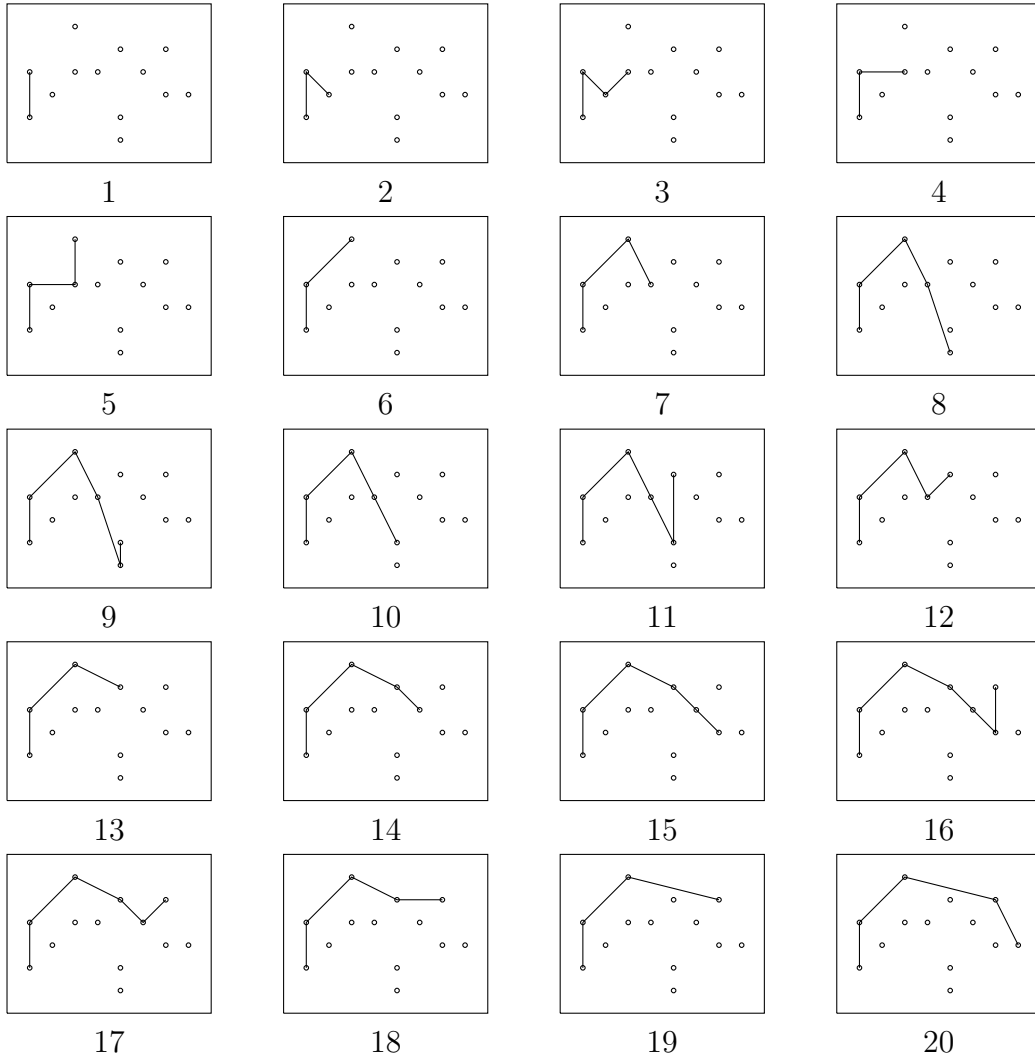
convex hull aşağıdaki gibi olur:



Andrew'in algoritması [3] belirli bir nokta kümesinden $O(n \log n)$ zamanda kolay bir şekilde convex hull oluşturmamızı sağlar. Algoritma ilk başta en sol ve en sağ noktaları bulunu convex hull 'i iki parçada yapar. İlk yukarı ve aşağı hull. İki parça da benzerdir o yüzden yukarı hull'u yapmakla odaklanabiliriz.

İlk başta noktaları öncelikli olarak x koordinatlarına göre sonra ikinci olarak y koordinatlarına göre sıralarız. Bundan sonra noktalardan geçeriz ve her noktayı hull'a ekleriz. Her bir noktayı hull'a eklediğimiz zaman hull'daki son çizginin sola dönmediğine emin oluruz. Sola döndüğü sürece hull'dan sondan ikinci noktayı sileriz.

Aşağıdaki resimler Andrew'in algoritmasının nasıl çalıştığını gösteriyor:



Kaynakça

- [1] A. V. Aho, J. E. Hopcroft and J. Ullman. *Data Structures and Algorithms*, Addison-Wesley, 1983.
- [2] R. K. Ahuja and J. B. Orlin. Distance directed augmenting path algorithms for maximum flow and parametric maximum flow problems. *Naval Research Logistics*, 38(3):413–430, 1991.
- [3] A. M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9(5):216–219, 1979.
- [4] B. Aspvall, M. F. Plass and R. E. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–123, 1979.
- [5] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.
- [6] M. Beck, E. Pine, W. Tarrat and K. Y. Jensen. New integer representations as the sum of three cubes. *Mathematics of Computation*, 76(259):1683–1690, 2007.
- [7] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Latin American Symposium on Theoretical Informatics*, 88–94, 2000.
- [8] J. Bentley. *Programming Pearls*. Addison-Wesley, 1999 (2nd edition).
- [9] J. Bentley and D. Wood. An optimal worst case algorithm for reporting intersections of rectangles. *IEEE Transactions on Computers*, C-29(7):571–577, 1980.
- [10] C. L. Bouton. Nim, a game with a complete mathematical theory. *Annals of Mathematics*, 3(1/4):35–39, 1901.
- [11] Croatian Open Competition in Informatics, <http://hsin.hr/coci/>
- [12] Codeforces: On "Mo's algorithm", <http://codeforces.com/blog/entry/20032>
- [13] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein. *Introduction to Algorithms*, MIT Press, 2009 (3rd edition).
- [14] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

- [15] K. Diks et al. *Looking for a Challenge? The Ultimate Problem Set from the University of Warsaw Programming Competitions*, University of Warsaw, 2012.
- [16] M. Dima and R. Ceterchi. Efficient range minimum queries using binary indexed trees. *Olympiad in Informatics*, 9(1):39–44, 2015.
- [17] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17(3):449–467, 1965.
- [18] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, 1972.
- [19] S. Even, A. Itai and A. Shamir. On the complexity of time table and multi-commodity flow problems. *16th Annual Symposium on Foundations of Computer Science*, 184–193, 1975.
- [20] D. Fanding. A faster algorithm for shortest-path – SPFA. *Journal of Southwest Jiaotong University*, 2, 1994.
- [21] P. M. Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 24(3):327–336, 1994.
- [22] J. Fischer and V. Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Annual Symposium on Combinatorial Pattern Matching*, 36–48, 2006.
- [23] R. W. Floyd Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.
- [24] L. R. Ford. Network flow theory. RAND Corporation, Santa Monica, California, 1956.
- [25] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8(3):399–404, 1956.
- [26] R. Freivalds. Probabilistic machines can use less running time. In *IFIP congress*, 839–842, 1977.
- [27] F. Le Gall. Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*, 296–303, 2014.
- [28] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 1979.
- [29] Google Code Jam Statistics (2017), <https://www.go-hero.net/jam/17>
- [30] A. Grønlund and S. Pettie. Threesomes, degenerates, and love triangles. In *Proceedings of the 55th Annual Symposium on Foundations of Computer Science*, 621–630, 2014.

- [31] P. M. Grundy. Mathematics and games. *Eureka*, 2(5):6–8, 1939.
- [32] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997.
- [33] S. Halim and F. Halim. *Competitive Programming 3: The New Lower Bound of Programming Contests*, 2013.
- [34] M. Held and R. M. Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10(1):196–210, 1962.
- [35] C. Hierholzer and C. Wiener. Über die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. *Mathematische Annalen*, 6(1), 30–32, 1873.
- [36] C. A. R. Hoare. Algorithm 64: Quicksort. *Communications of the ACM*, 4(7):321, 1961.
- [37] C. A. R. Hoare. Algorithm 65: Find. *Communications of the ACM*, 4(7):321–322, 1961.
- [38] J. E. Hopcroft and J. D. Ullman. A linear list merging algorithm. Technical report, Cornell University, 1971.
- [39] E. Horowitz and S. Sahni. Computing partitions with applications to the knapsack problem. *Journal of the ACM*, 21(2):277–292, 1974.
- [40] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.
- [41] The International Olympiad in Informatics Syllabus, <https://people.ksp.sk/~misof/ioi-syllabus/>
- [42] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.
- [43] P. W. Kasteleyn. The statistics of dimers on a lattice: I. The number of dimer arrangements on a quadratic lattice. *Physica*, 27(12):1209–1225, 1961.
- [44] C. Kent, G. M. Landau and M. Ziv-Ukelson. On the complexity of sparse exon assembly. *Journal of Computational Biology*, 13(5):1013–1027, 2006.
- [45] J. Kleinberg and É. Tardos. *Algorithm Design*, Pearson, 2005.
- [46] D. E. Knuth. *The Art of Computer Programming. Volume 2: Seminumerical Algorithms*, Addison–Wesley, 1998 (3rd edition).
- [47] D. E. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching*, Addison–Wesley, 1998 (2nd edition).

- [48] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.
- [49] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics doklady*, 10(8):707–710, 1966.
- [50] M. G. Main and R. J. Lorentz. An $O(n \log n)$ algorithm for finding all repetitions in a string. *Journal of Algorithms*, 5(3):422–432, 1984.
- [51] J. Pachocki and J. Radoszewski. Where to use and how not to use polynomial string hashing. *Olympiads in Informatics*, 7(1):90–100, 2013.
- [52] I. Parberry. An efficient algorithm for the Knight’s tour problem. *Discrete Applied Mathematics*, 73(3):251–260, 1997.
- [53] D. Pearson. A polynomial-time algorithm for the change-making problem. *Operations Research Letters*, 33(3):231–234, 2005.
- [54] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6):1389–1401, 1957.
- [55] 27-Queens Puzzle: Massively Parallel Enumeration and Solution Counting. <https://github.com/preusser/q27>
- [56] M. I. Shamos and D. Hoey. Closest-point problems. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, 151–162, 1975.
- [57] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7(1):67–72, 1981.
- [58] S. S. Skiena. *The Algorithm Design Manual*, Springer, 2008 (2nd edition).
- [59] S. S. Skiena and M. A. Revilla. *Programming Challenges: The Programming Contest Training Manual*, Springer, 2003.
- [60] SZKOpuł, <https://szkopul.edu.pl/>
- [61] R. Sprague. Über mathematische Kampfspiele. *Tohoku Mathematical Journal*, 41:438–444, 1935.
- [62] P. Stańczyk. *Algorytmika praktyczna w konkursach Informatycznych*, MSc thesis, University of Warsaw, 2006.
- [63] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.
- [64] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.
- [65] R. E. Tarjan. Applications of path compression on balanced trees. *Journal of the ACM*, 26(4):690–715, 1979.

- [66] R. E. Tarjan and U. Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. In *Proceedings of the 25th Annual Symposium on Foundations of Computer Science*, 12–20, 1984.
- [67] H. N. V. Temperley and M. E. Fisher. Dimer problem in statistical mechanics – an exact result. *Philosophical Magazine*, 6(68):1061–1063, 1961.
- [68] USA Computing Olympiad, <http://www.usaco.org/>
- [69] H. C. von Warnsdorf. *Des Rösselsprunges einfachste und allgemeinste Lösung*. Schmalkalden, 1823.
- [70] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.

Dizin

- 2SAT problemi, 168
- 2SUM problemi, 84
- 3SAT problemi, 170
- 3SUM Problemi, 85
- akış, 191
- alfabe, 255
- alt ağaç, 141
- alt dize, 255
- alt dizi, 255
- alt küme, 53
- altküme, 12
- amortize analizi, 83
- ancak ve ancak, 13
- Andrew'in Algoritması, 291
- anti zincir, 203
- aralarında asal, 211
- aralık sorgusu, 89
- ardıl yollar, 162
- aritmetik ilerleme, 11
- asal, 207
- asal çarpanlarına ayırma, 207
- ata, 171
- Atın turları, 189
- ayakkabı bağı formülü, 283
- açgözlü algoritma, 63
- ağaç, 116, 141
- ağaç dolaşma dizisi, 172
- ağaç sorgusu, 171
- ağırlıklı çizge, 117
- basit çizge, 118
- bağlı çizge, 116, 127
- bağımsız grup, 200
- bağımsızlık, 240
- beklenen değer, 240
- Bellman–Ford algoritması, 131
- binary kod, 68
- Binet'in formülü, 14
- binom dağılımı, 241
- binom katsayısı, 218
- bipartite çizge, 118, 128
- birim matris, 230
- birleşim, 12
- birleştirme sıralama, 29
- bit gösterimi, 101
- bit kaydırması, 103
- bitset, 46
- boyama, 118, 245
- Burnside'in Lemma'sı, 224
- bölen, 207
- bölünebilirlik, 207
- Cayley'in Formülü, 225
- dağılım, 241
- De Bruijn dizisi, 188
- deque, 47
- derece, 117
- derinlik öncelikli arama, 123
- determinant, 231
- değil, 13
- değişilik farkı, 80
- Dijkstra'nın Algoritması, 161
- Dijkstra'nın algoritması, 134
- Dilworth'un Teoremi, 203
- dinamik dizi, 39
- dinamik programlama, 71
- Dinamik Segment Ağaçları, 273
- Dirac'ın Teoremi, 187
- Diyonfantus eşitliği, 214
- doğal logaritma, 15
- doğum günü paradoksu, 259
- döngü, 115, 127, 157, 163
- döngü bulma, 163
- düzensizlik, 223

düğüm, 115
 düğüm kaplaması, 199
 dış derece, 117
 ebeveyn, 141
 Edmonds–Karp Algoritması, 194
 en büyük kapsayan ağaç, 150
 en büyük ortak bölen, 210
 en küçük kapsayan ağaç, 149
 en küçük ortak kat, 210
 en kısa yol, 131
 en uzun artan altdizi, 76
 en yakın küçük elemanlar, 85
 en yakın ortak ata, 175
 en yakın çift, 289
 Eratosten Kalburu, 210
 Euler Devresi (Euler devresi), 184
 Euler yolu, 183
 Euler’in Teoremi, 212
 Euler’in totient fonksiyonu, 211
 evrensel küme, 12
 eşleşme, 197
 faktöriyel, 14
 fark, 12
 fark dizisi, 99
 Faulhaber’in Formülü, 10
 Fenwick ağacı, 92
 Fermat’ın Teoremi, 212
 Fibonacci sayısı, 14, 216, 232
 Floyd’un algoritması, 164
 Floyd–Warshall Algoritması, 137
 fonksiyonel çizge, 162
 Ford–Fulkerson Algoritması, 192
 Freivalds’ın algoritması, 244
 Genişletilmiş Öklid Algoritması, 214
 genişlik öncelikli arama, 125
 geometri, 277
 geometrik dağılım, 242
 geometrik ilerleme, 11
 geri izleme, 56
 girdi ve çıktı, 4
 Goldbach’ın Konjektürü, 209
 Grundy sayısı, 251
 Grundy’nin Oyunu, 253
 güçlü bağlanmış parçalar, 165

güçlü bağlanmış çizge, 165
 Hall’ın Teoremi, 199
 Hamilton Devresi, 187
 Hamilton yolu, 187
 Hamming mesafesi, 106
 harmonik toplam, 11
 harmonik toplamı, 210
 hash, 257
 hash değeri, 257
 heap, 48
 Heron Formülü, 277
 heuristic, 189
 Hierholzer’in algoritması, 185
 Huffman kodlaması, 69
 hızlı seçme, 244
 hızlı sıralama, 244
 iki işaretçi methodu, 83
 iki-boyutlu segment ağacı, 276
 ikili arama, 34
 ikili ağaç, 147
 ikili indisli ağaç, 92
 indis sıkıştırması, 99
 ise, 13
 iteratör, 43
 iç derece, 117
 içirme-dışarma, 222
 kabarcık sıralaması, 27
 Kadane’in Algoritması, 25
 kalan, 7
 kapsayan ağaç, 149, 236
 kare matris, 229
 karekök algoritması, 263
 karmaşık, 278
 karmaşık sayı, 278
 karmaşıklık sınıfları, 22
 karşılaştırma fonksiyonu, 34
 karşılaştırma işlemi, 33
 Katalan sayısı, 220
 kayan noktalı sayı, 7
 kaybeden durum, 247
 kazanan durum, 247
 kenar, 115
 kenar listesi, 121
 kesim, 192

kesişim, 12
kesişim noktası, 288
Kirchhoff'un Teoremi, 236
knapsack, 78
kodlama dili, 3
kofaktör, 231
kombinatorik, 217
komşu, 117
komşuluk listesi, 119
komşuluk matrisi, 120
Kosaraju'nun Algoritması, 166
koşullu olasılık, 239
kraliçe problemi, 56
Kruskal'ın Algoritması, 150
kuyruk, 48
kök, 141
kök ortada, 147
kök sonda, 147
köklü ağaç, 141
kübik algoritma, 22
küme, 12, 41
küme teorisi, 12
König'in Teoremi, 199

Lagrange'in Teoremi, 215
Laplas Matrisi, 236
Las Vegas algoritması, 243
lazy propagation, 270
Legendre'nin Konjektürü, 209
Levenshtein farkı, 80
linear algoritma, 22
linear tekrar, 232
logaritma, 14
logaritmik algoritma, 22

makro, 9
maksimum akış, 191
maksimum alt dizi toplamı, 23
maksimum bağımsız grup, 200
maksimum sorgusu, 89
Manhattan uzaklığı, 284
mantık, 13
map, 42
Markov Zinciri, 242
matris, 229
matris kuvveti, 231
matris çarpımı, 230, 244

maximum eşleşme, 197
memoization, 73
mex fonksiyonu, 251
minimum düğüm kaplaması, 199
minimum kesim, 192, 195
minimum sorgusu, 89
misère oyunu, 250
Mo'nun Algoritması, 267
modüler aritmetik, 7, 211
modüler çarpımsal ters, 213
Monte Carlo Algoritması, 243
mükemmel eşleşme, 199
mükemmel sayı, 208

negatif döngü, 133
next_permutation, 55
niceleyici, 13
nim oyunu, 249
nim toplamı, 249
nokta, 278
NP-hard problem, 23

olasılık, 237
Ore'nin Teoremi, 187
ortada buluşmak, 60

çift, 33
parentez gösterimi, 221
parça, 116
parça çizgesi, 165
Pascal'ın Üçgeni, 219
periyot, 255
permütasyon, 55
Pick'in Teoremi, 284
Pisagor üçlüsü, 216
polinom algoritma, 23
polinom hashleme, 257
predicate, 13
prefix toplam dizisi, 90
Prim'in Algoritması, 155
priority queue, 48
Prüfer kodu, 226

quadratic algoritma, 22

random_shuffle, 43
rastgele algoritma, 243
rastgele değişken, 240

reverse, 43
 Rotasyon, 255

 sabit faktör, 23
 sabit zamanlı algoritma, 22
 sayarak sıralama, 30
 sayılar teorisi, 207
 segment Ağacı, 95
 segment ağacı, 269
 seyrek segment ağaç, 273
 son ek, 255
 sort, 43
 sıralama, 32
 sparse table, 91
 SPFA algoritması, 134
 Sprague–Grundy Teoremi, 250
 string, 40
 sözlük sıralaması, 256
 sürekli dağılım, 241
 sürekli segment ağaç, 274
 sürgülü pencere, 87
 sürgülü pencere minimumu, 87
 sınır, 256
 sıra istatistiği, 244
 sıradan çizge, 117
 sıralama, 27

 tam sayı, 6
 tam çizge, 117
 tamlayan, 12
 tembel segment ağaç, 270
 ters elemanlar, 28
 ters matris, 232
 ters operasyonu, 103
 toplam sorgusu, 89
 topolojik sıralama, 157
 transpoz, 229
 trie, 256
 tuple, 33
 typedef, 8
 twin prime, 209

 union-find yapısı, 153
 uzaklık fonksiyonları, 284

 ve, 13
 ve operasyonu, 102
 vektör, 39, 229, 278
 vektörel çarpım, 280
 veri sıkıştırma, 68
 veri yapısı, 39
 veya, 13
 veya operasyonu, 102

 Warnsdorf’un kuralı, 189
 Wilson’un Teoremi, 216

 xor operasyonu, 103

 yaprak, 141
 yazı, 255
 yazı hashleme, 257
 yol, 115
 yol kaplaması, 200
 yönlü çizge, 116
 yığın, 47

 Z-algoritması, 259
 Z-dizisi, 259
 zaman karmaşıklığı, 19
 Zeckendorf’in Teoremi, 216

 Çin Kalan Teoremi, 215
 Öklid uzaklığı, 284
 Öklid’in Algoritması, 210
 Öklid’in formülü, 216
 Ölçekleme Algoritması, 195
 çap, 143
 çarpan, 207
 çarpışma, 258
 çizge, 115
 çizgi kesişimi, 281
 çizgi süpürme, 287
 çocuk, 141
 çok terimli katsayı, 220
 ön ek, 255
 önce kök, 147
 örüntü bulma, 255

 şifre, 68