VERİ YAPILARI VE ALGORİTMALAR 1

Mustafa EGE

DERS NOTLARI

2008-2009 Güz Dönemi

by **Muhammed DEMİRBAŞ**

BÖLÜM 0 – DERSE GİRİŞ – (kitaplarda yazmaz!)

Herkesin elinde referans olarak sürekli bulunması gereken bir kitap, C dilinin kurucusu Dennis Ritchie tarafından yazılmış: ***The C Programming Language,*** *Dennis M. Ritchie – Brain N. Kerninghan*

Derste takip edilecek kitap: ***Fundamentals of Data Structures in C,*** *Ellis Horowitz, Sartaj Sahni, Susan Anderson-Freed*. Bu kitap ilk olarak *Fundamentals of Data Structures* ismiyle yazılmış olup uygulamalar SPARKS dili üzerinde gerçekleştirilmişti. (Bu versiyonun eBook hali internette dolaşıyor) Daha sonra C dili üzerinde, en son da C++ üzerinde gerçekleştirilmiş halleri piyasaya çıktı. Kodların yazıldığı diller dışında kitabın içeriğinde önemli bir değişiklik yok, hemen hemen aynı. Bununla birlikte en kapsamlısı C++ üzerine yazılmış olan. Biz dersimizde C dili üzerinde yazılmış olan kitabı kullanacağız.

BÖLÜM 1 – TEMELLER – (s. 1)

SİSTEM YAŞAM DÖNGÜSÜ (System Life Cycle) – (s. 1)

Genel olarak yazılımlar sistem yaşam döngüsü (system-cycle) adı verilen bir süreç içinden geçerler. Bu sürecin aşamaları aşağıdaki gibi tanımlanabilir.

1) **Gereksinimler (Requirements):** Programlamayla ilgili tüm projeler, projenin amacını belirten şartların tanımlanmasıyla başlar. Gereksinmeler, programcıya verilen girdiler ve bu girdiler sonucu üretilmesi gereken çıktıların ne olması gerektiği sorusuyla tanımlanır.

2) **Çözümleme (Analysis):** Sistemin gereksinimleri belirlendikten sonra ilk yapılacak işlerden biri, problemi analiz etmektir. İki yol vardır:

   i. *Aşağıdan yukarı (bottom-up) analiz:* Problem parçalara bölünür ve her bir parçanın üzerinde odaklaşılır. Proje üzerinde tecrübe sahibi isek bu yolu tercih edebiliriz. Direk modülleri görerek problemi modül modül çözümleriz.

   ii. *Yukarıdan aşağıya (top-down) analiz:* Proje konusunda bir master plan yapılır. "Projenin amacı nedir?", "Son ürün ne olacak?" gibi sorularla problem çözümü yönetilebilir. Problem alt parçalara bölünür. Bu safhada çeşitli teknikler söz konusudur. Daha fazla zaman harcanır. Fakat geriye dönüşler ve hataya düşme olasılığı daha azdır. Proje hakkında tecrübe sahibi değilsek bu yolu tercih edebiliriz.

3) **Tasarım (Design):** Tasarıma ayırdığımız vakit arttıkça hata oranı azalacaktır. Bu aşama, analiz aşamasındaki çalışmaların devamıdır. Tasarımcı veri objelerini ve objeler arasında bulunan ilişkileri bu safhada tanımlar. Veri objeleri, soyut veri türü tanımlanmasını, işlemler ise algoritmanın tanımlanmasını gerektirir. Her ikisi de programlama dilinden bağımsızdırlar (*language independent*). Örneğin bir öğrenci veri kütüğünün içermesi gereken veri öğelerini belirleriz. Fakat bu kütük için belirli bir gerçekleştirimi yapmamış olabiliriz. Diğer bir deyişle, kodlama ayrıntılarını vermemiş olabiliriz. (Dizi, bağlaçlı liste, ağaç veri yapısı…) Gerçekleştirimi ertelemek suretiyle daha etkili bir gerçekleştirimi seçme fırsatı yakalamamız mümkündür.

4) **İnceltme ve kodlama (Refinement and Coding):** Belirlenen veri yapıları üzerinde işlem yapacak algoritmaları bu aşamada kodlarız. "Veri objeleri gösterimi", algoritmaların etkinliğini belirlemede önemli rol oynayabilir. Benzer bir projede çalışmış bir arkadaşımızla yapacağımız sohbet veya ürettiğimiz alternatif çözümlerden biri, çözüm için en iyi yaklaşımı verebilir.

> Kodlamayı ne kadar geciktirirsen, arka planda o kadar iyi bir metot yakalayabilirsin.

5) **Doğrulama (Verification):** Bu safhada, geliştirilen programın doğruluğunun ispatı, geniş bir veri grubu üzerinde test etme ve hatalardan arındırma işlemleri gerçekleştirilir. Bunların her biri bir araştırma konusudur.

   i. **Doğruluk ispatı (Correctness Proofs):** Matematikte bazı teknikleri kullanarak programın doğruluğu ispatlanabilir. Fakat bu işlem büyük projeler için hem zordur, hem de zaman alıcıdır. Büyük projeler için baştan sona bir ispat geliştirmek zaman kısıtlayıcısı nedeniyle neredeyse imkansızdır. Daha önceden doğruluğu ispatlanmış algoritmaları kullanmak, hata sayısını azaltabilir.

   ii. **Sınama (Testing):** Algoritmayı bir programlama diliyle kodlama ihtiyacı yok iken, kodlama safhası öncesi ve kodlama safhası sırasında doğrulama ispatlarını yapabiliriz. Fakat test etme işlemi için çalışan bir kod ve test verisine bu aşamada gereksinim vardır. Test verisi, programa ait tüm olası senaryoları içerecek biçimde hazırlanmaya çalışılır. Acemi programcılar, programı sözdizimi hatası

(syntax error) vermeden çalışmışsa, programın doğru olduğunu zannederler. İyi bir test verisi, programın her kesiminin doğru olarak çalıştığını onaylamalıdır. Bir programın hatadan arındırılımış (error-free) bir program olmasının yanı sıra programın işletim zamanı (running-time) da önemlidir. Hatadan arındırılmış, fakat yavaş çalışan bir programın da fazla değeri yoktur.

iii. **Hata giderme (Error Removal):** Doğruluk ispatı ve sistem testleri hatalı kod ile uğraştığımıza işaret ederse tasarım ve kodlama kararlarına bağlı olarak bu hataları yok edebiliriz.

SEÇMELİ SIRALAMA (Selection Sort) – (s. 5)

Sıralı olmayan değerler içeren bir diziyi sıralama algoritması (s. 5):

```
for( i=0 ; i<n ; i++ ){
      examine list[i] to list[n-1] and suppose that
      the smallest integer is at list[min];
      interchange list[i] and list[min];
}
```

Yazılmış algoritmaları beğenmeyip daha iyi alternatifler sunabilecek seviyeye gelmeniz lazım.

Bu algoritmanın C dili ile gerçekleştirimi (s. 7):

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 101
#define SWAP(x,y,t) ((t)=(x), (x)=(y), (y)=(t))

void sort( int[], int );

int main( void ){
    int i, n;
    int list[MAX_SIZE];

    printf("Dizi Boyunu Giriniz: ");
    scanf("%d", &n);

    if( n<1 || n>MAX_SIZE ){
        printf("Hatali n degeri!\n");
        exit(1);
    }

    for( i=0 ; i<n ; i++ ){      // Diziye değerleri ata
        list[i] = rand()%100;    // [0,99] aralığından bir değer
        printf("%d ", list[i]);
    }

    printf("\n\nSirali Degerler:\n");
    sort(list, n);               // Sırala
    for( i=0 ; i<n ; i++ )
        printf("%d ", list[i]);

    return 0;
} // --> end main()

void sort( int list[], int n ){
    int i, j, min, temp;
    for( i=0 ; i<n ; i++ ){
        min = i;
        for( j=0 ; j<n ; j++ )
            if( list[j] > list[min] ){
                min = j;
                SWAP( list[i], list[min], temp );
            }
    }
} // -> end sort()
```

Programda SWAP makrosu kullanıldı. Bu makronun eşdeğeri olan bir fonksiyon da kullanılabilirdi (s. 6):

```
void SWAP( int *x, int *y ){
    int temp = *x;
    *x = *y;
    *y = temp;
} // -> end SWAP()
```

Bu durumda `sort()` fonksiyonu içindeki satırı, şu şekilde değiştiririz ve artık `temp` değişkeni gereksiz olur:

```
                SWAP( &list[i], &list[min] );
```

ÖZYİNELİ ALGORİTMALAR (Recursive Algorithms)

Bütündeki çözüm mantığı alt problemlerde de geçerli ise özyinelemeli algoritmalar kullanılabilir.

Fonksiyonlar kendi kendilerini çağırabilecekleri gibi (direct recursion), çağırdıkları bir fonksiyon tarafından da çağrılabilirler (indirect recursion). Özyineli algoritmalar hem güçlü algoritmalardır, hem de karmaşık yapıları daha rahat açıklayabilirler.

> *"Hocam bu özyinelemeli çözüm çok bellek yemez mi?"* Sakın ha! Önce kodu yazmayı öğrenin, algoritma karmaşıklığını hesaplayabiliyorsanız o zaman böyle diyebilirsiniz. Siz daha özyineli kod yazmayı öğrenmediniz ki!

İKİLİ ARAMA (Binary Search) – (s. 6)

Sıralı değerler içeren bir dizide bir değer aramak istiyoruz. Aranan değer dizi içinde bulunursa indis değerini, bulunamazsa -1 değerini döndürecek bir yordam yazacağız. Binary search (ikili arama) algoritmasını kullanacağız.

İteratif (Döngüsel) Çözüm [$O(\lg(n))$] – (s. 9)

```
#define COMPARE(x,y) ( (x)<(y) ? -1 : (y)<(x) ? 1 : 0 )

int binsearch( int list [], int searchnum, int left, int right ){
    int middle;
    while( left <= right ){
        middle = ( left + right ) / 2;
        switch( COMPARE( list[middle], searchnum ) ){
            case -1: left = middle + 1; break;
            case  0: return middle;
            case  1: right = middle -1;
        }
    }
    return -1;
}
```

> list[i]'yi değil, i'yi döndür. Neden? Hata değeri olan -1 ile dizi elemanı olabilecek bir -1 karışmasın diye.

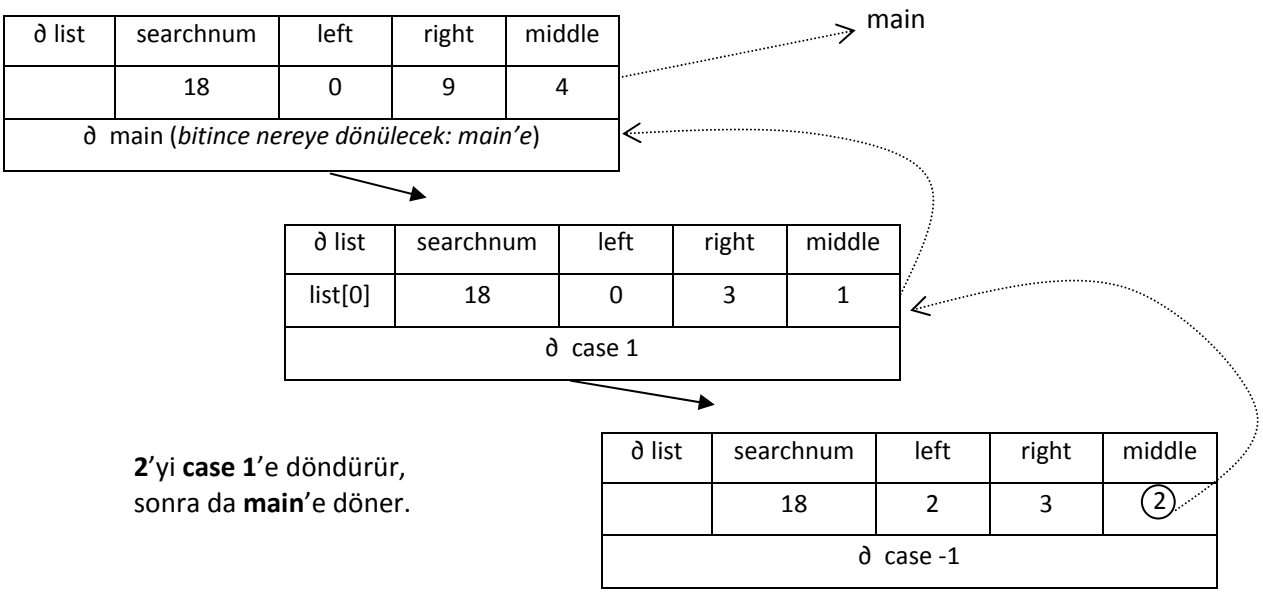
Rekürsif (Özyinelemeli) Çözüm – (s. 11)

```
int binsearch( int list [], int searchnum, int left, int right ){
    int middle;
    while( left <= right ){
        middle = ( left + right ) / 2;
        switch( COMPARE( list[middle], searchnum ) ){
            case -1: return binsearch( list, searchnum, middle+1, right );
            case  0: return middle;
            case  1: return binsearch( list, searchnum, left, middle-1 );
        }
    }
    return -1;
}
```

Stack Activation Frame: Bir programın işletimi sırasında bir fonksiyon çağrıldığında özel bir yığıt kullanılır. Bu yığıtın adı sistem yığıtıdır. Denetim çağrıları fonksiyona geçmeden önce bir hazırlık aşaması vardır. Bu aşamada yapılanlar fonksiyondan nereye dönüleceği, fonksiyona aktarılan parametrelerin saklanması,

parametrelerin kullanımı gibi konulara bir açıklık getirir. Bu amaç için düzenlenen yapıya iletişim-denetim yapısı yada **yığıt çerçevesi** adı verilir. Bir program denetimi ele aldığında sistem yığıtının başlangıçta iki değer içerdiği gözlenir: dönüş adresi ve önceki yığıt çerçevesine gösterge. Eğer ana yordam (main) içinde bir fonksiyon çağırma söz konusu ise yerel (local) değişkenler ve çağıran yordamın parametreleri yığıt çerçevesine eklenir. Daha sonra fonksiyondan nereye dönüleceği bilgisi ve önceki yığıt çerçevesi göstergesi sistem yığıtına eklenir.

İkili aramada örnek test verisi olarak kullanılan dizi									
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
4	**13**	**18**	**19**	**24**	**27**	**33**	**36**	**39**	**40**

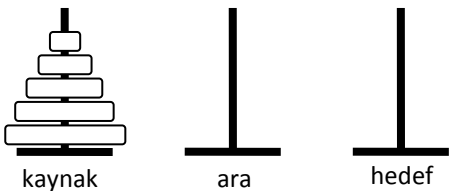


2'yi **case 1**'e döndürür, sonra da **main**'e döner.

HANOİ KULELERİ PROBLEMİ – (s. 14/11)

Oyun tanımı: Üç tane kule var: kaynak, ara, hedef. Kaynak kulede *n* tane disk bulunuyor. Her bir disk, kendi üzerindekinden daha büyük boyuttadır. *Oyunun amacı* kaynak kuledeki tüm diskleri hedef kuleye taşımaktır. Taşıma sırasıda ara kuleden yardım alınabilir. Fakat;

1. Her seferde tek bir disk taşınabilir.
2. Bir disk, kendisinden daha küçük bir diskin üzerine konamaz.



Özyinelemeli Çözüm: En büyük diski hedef kuleye taşıyabilmek için üstündeki *n-1* tane diski ara kuleye taşıdığımız bir an bulunmalıdır. Öyleyse problemdeki özyineli ilişki (*recursive relation*) şöyle yazılır: $h(n) = h(n-1) + 1 + h(n-1)$

```
void hanoi( int n, char *kaynak, char *ara, char *hedef ){
    if( n==1 ) printf("%s -> %s\n", kaynak, hedef);
    else{
        hanoi( n-1, kaynak, hedef, ara );
        printf("%s -> %s\n", kaynak, hedef);
        hanoi( n-1, ara, kaynak, hedef );
    }
}

int main(){
    hanoi( 3, "A", "B", "C");
    return 0;
}
```

Öyle problemler var ki, ancak rekürsif yolla çözülebilir.

ÖDEV: `hanoi( 3, "1", "2", "3");` özyinelemeli yordam çağrısı için yığıt çerçevesi gösterimini çiziniz.

∂ ana			
n	kaynak	ara	amaç
3	1	2	3

PERMÜTASYON PROBLEMİ [$O(n!)$] – (s. 12)

Eleman sayısı $n \geq 1$ olmak üzere bir küme verildiğinde söz konusu kümenin olası tüm permütasyonlarını yazdırmak istiyoruz. Küme n eleman içerdiği için $n!$ dizilişin olduğunu biliyoruz. Örneğin $\{a,b,c\}$ kümesi verildiğinde olası dizilimler; $\{a,b,c\}$, $\{a,c,b\}$, $\{b,a,c\}$, $\{b,c,a\}$, $\{c,a,b\}$, $\{c,b,a\}$ olur.

Küme 4 elemanlı $\{a,b,c,d\}$ kümesi olsaydı olası dizilişler;

- a'yı takip eden $\{b,c,d\}$ kümesinin tüm olası dizilişleri,
- b'yi takip eden $\{a,c,d\}$ kümesinin tüm olası dizilişleri,
- c'yi takip eden $\{a,b,d\}$ kümesinin tüm olası dizilişleri,
- d'yi takip eden $\{a,b,c\}$ kümesinin tüm olası dizilişlerinin toplamı olurdu.

```
void perm( char *list, int i, int n ){
    int j, temp;
    if( i==n ){
        for( j=0 ; j<n ; j++ )
            printf("%c", list[j]);
        printf("\n");
    }
    else {
        for( j=i ; j<n ; j++ ){
            SWAP(list[i], list[j], temp );
            perm( list, i+1, n );
            SWAP(list[i], list[j], temp );
        }
    }
} // --> end perm()
```

TABLOLAR VAR... TABLOLAR VAR... TABLOLAR VAR... TABLOLAR VAR... TABLOLAR VAR... TABLOLAR VAR...
TABLOLAR VAR... TABLOLAR VAR... TABLOLAR VAR... TABLOLAR VAR... TABLOLAR VAR... TABLOLAR VAR...
TABLOLAR VAR... TABLOLAR VAR... TABLOLAR VAR... TABLOLAR VAR... TABLOLAR VAR... TABLOLAR VAR...
TABLOLAR VAR... TABLOLAR VAR... TABLOLAR VAR... TABLOLAR VAR... TABLOLAR VAR... TABLOLAR VAR...
TABLOLAR VAR...

ÖDEV SORULARI

1) $n!$ değerini bulan fonksiyonu özyineli olarak yazınız.

2) Fibonacci serisi $\rightarrow f_0 = 0 \quad f_1 = 1 \quad f_i = f_{i-1} + f_{i-2} \quad (i > 1)$ ise, f_i değerini bulan özyineli yordamı yazınız.

3) $\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1}$ verildiğine göre $\binom{n}{m} = \frac{n!}{m! \cdot (n-m)!}$ değerini hesaplayan özyineli yordamı yazınız.

ÖDEV CEVAPLARI

```
long fakt( long number ){
    if( number <= 1 )
        return 1;
    else
        return number * fakt(number-1);
}
```

```
long fibo( long deger ){
    if( deger <= 1 )    // if( deger == 0 || deger == 1 )
        return deger;
    else
        return fibo(deger-1) * fibo(deger-2);
}
```

```
long komb(){
    if( n==m || m==0 )
        return 1;
    else
        return komb(n-1, m) + komb(n-1, m-1);
}
```

PERFORMANS ANALİZİ (Performance Analysis) – (s. 18)

Bir programını "işlenebilir", "kabul edilebiler" ve "kayda değer" olduğunu belirtmek için aşağıdaki soruların cevaplarının araştırılması gerekir.

1- Program, kendisinden istenenleri karşılıyor mu?

2- Doğru olarak çalışıyor mu?

3- Nasıl işletileceği ve nasıl kullanılacağı konusunda belgelendirmeye sahip mi? (Açıklama satırları da dahil)

4- TRUE, FALSE mantıksal değerlerini oluşturmada fonksiyonları etkili bir biçimde kullanabiliyor mu?

5- Program kodu okunabilir (*readable*) mi?

6- Program, ana (*primary*) ve ikincil (*secondary*) belleği etkili (*efficent*) bir biçimde kullanabiliyor mu?

7- Programın işletim zamanı (*running-time*) kabul edilebilir mi?

Son iki kriter, ***performans değerlendirme*** (*performance evolution*) üzerinde odaklaşır. Performans değerlendirmenin iki ayağı vardır:

1. **Performans analizi**: Makineden bağımsız olarak zaman ve bellek ile ilgili tahminler yapılır.
2. **Performans ölçümü**: Bilgisayara bağımlı işletim zamanı elde edilir.

BELLEK KARMAŞIKLIĞI (Space Complexity) – (s. 19)

Bellek karmaşıklığı, bir programın işletimini tamamlaması için ihtiyaç duyduğu bellek miktarıdır. İki bileşeni vardır:

```
Her şey kısıtlı imiş gibi kod
yazmalısınız. Bellek bol nasıl olsa
diye boşa bellek harcayamazsınız.
```

1- **Sabit Bellek Gereksinimi (Fixed-Space Requirement)**: Programın girdi ve çıktı büyüklüğüne bağlı değildir. Kodun yüklenmesi için gereken belleği, sabit uzunluklu değişkenler ve sabit değerler için gereken belleği içerir.

2- **Değişken Bellek Gereksinimi (Variable-Space Requirement):** I ile sembolize edebileceğimiz bir girdi büyüklüğüne bağlı olarak değişkenlerin gerek duyduğu bellek miktarını içerir. I girdisi üzerinde çalışan P programı için değişken bellek gereksinimi ile $S_p(I)$ gösterilir. Toplam bellek gereksinimi $S_p(I)+c=S_p$ olacaktır. Örneğin şu program için yalnızca sabit bellek gereksinimi söz konusudur, $S_{abc}(I)=0$'dır.

```
float abc( float a, float b, float c ){
    return a + b + b*c - (a+b-c)/(a+b) + 4.00;
}
```

```
float sum( float list[], int n ){
    float tempsum = 0;
    int i;
    for( i=0 ; i<n ; i++ )
        tempsum += list[i];
    return tempsum;
}
```

sum yordamına list dizisinin ilk elemanının adresi gönderildiği için $S_{sum}(n) = 0$, (girdi karakteristiği ile ilgili sum yordamında yalnızca boyut bilgisi (n) kayda değer olduğu için I yerine n yazdık. Eğer sum yordamına list dizisinin elemanlarının hepsi *by value* yaklaşım ile oluşturulmuş olsaydı $S_{sum}(I) = S_{sum}(n) = n$ olurdu.

Bir programın bellek karmaşıklığı çözümlemesinde genelde *değişken bellek gereksinimi* ile ilgilenilir.

> Performansı yüksek programlar yazmak neden bu kadar önemli? Bir savaş uçağının bombalama sistemini siz programlıyorsunuz. Savaşta sizin yazdığınız program 3 sn'de bomba bırakıyor da karşı tarafınki 2 sn'de bırakıyor. Savaşı baştan kaybettiniz demektir. "Ben savaşmıyorum, kabul etmiyorum." mu diyeceksiniz?

```
float rsum( float list[], int n ){
    if( n )
        return rsum( list, n-1 ) + list[n-1];
    return 0;
}
```

2 byte (list) + 2 byte (n) + 2 byte (dönüş adresi) = en az 6 byte. $n = 1000$ için $6 \cdot 1000 = 6000$ byte'lık bir bellek kaybı olduğunu bilmemiz gerekir. (far türü bir bellek değil, near türü bir bellek)

ZAMAN KARMAŞIKLIĞI (Time Complexity) – (s. 21)

> Zaman, bellekten daha önemli!

Bir P programının kullandığı zaman $T(P)$, hem derleme zamanını (compile-time), hem de işletim zamanını (run-time / execute-time) içerir. Daha çok işletim zamanı ile ilgileniriz. $T(P)$ değerini bulmak için derleyicinin niteliklerini bilmek zorunlu olabilir.

Elimizde yalnızca toplama ve çıkarma işlemi yapan bir program olduğunu varsayalım. n girdi niteliğini göstermek üzere $T(P) = c_a \cdot ADD(n) + c_s \cdot SUB(n) + c_l \cdot LDA(n) + c_{st} \cdot STA(n)$ biçiminde açıklanabilir. c değerleri her bir işlemin ne kadarlık bir sürede gerçekleştiğine işaret eden sabit değerlerdir. ADD, SUB, LDA ve STA ise addition, subtraction, load ve store işlemlerinin sayısıdır. İşletim zamanını belirlemede en iyi yaklaşım sistem clock değerini kullanmaktır. Buna alternatif olarak programdaki adımların sayısı ile de ilgileniriz. Bu yaklaşım bilgisayardan bağımsız bir çalışma (*machine-independent*) yapma fırsatı verir.

```
float sum( float list[], int n ){
    float tempsum = 0;
    int i;
    for( i=0 ; i<n ; i++ )          n+1
        tempsum += list[i];         n
    return tempsum;                 1
}                          toplam:  2n+3 adım
```

Fonksiyonumuzu, işletilen satırları sayacak şekilde yeniden yazalım:

```
float sum( float list[], int n ){
    float tempsum = 0;  count++;
    int i;
    for( i=0 ; i<n ; i++ ){
        count++;
        tempsum += list[i];  count++;
    }
    count++; // last execution of for loop
    count++; return tempsum;
}
```

```
float rsum( float list[], int n ){
    if( n )                                      n+1
        return rsum( list, n-1 ) + list[n-1];    n
    return 0;                                    1
}                                      toplam:   2n+2
```

```
void add( int a[][MAX_SIZE], int b[][MAX_SIZE], int c[][MAX_SIZE], int rows, int cols ){
    int i, j;
    for( i=0 ; i<rows ; i++ )              rows+1
        for(j=0 ; j<cols ; j++)            rows*(cols+1)
            c[i][j] = a[i][j] + b[i][j];   rows*cols
}                              toplam:     2*rows*cols + 2*rows + 1
```

NOT: Kitabın 30. sayfasını inceleyebilirsiniz (Asymptotic Notation).

NOT: Matris çarpımının algoritma karmaşıklığı $O(n^3)$'tür (s. 23, 24, 25, 33).

ASİMPTOTİK GÖSTERİM (Asymptotic Notation) – (s. 30)

> Bir arkadaş da sınav sorusunu O(n)'de çözmüş. Sonuna da m kez dönen boş bir döngü koymuş. Hoca "O(n+m)'de çözün." dedi ya! ☺ Gerek yok. Dersimizde zaten etkili kod yazmayı öğretiyoruz. Hoca O(n+m) demiş, sen daha iyisini yapabiliyorsan ne alâ!

Performans değerlendirmede zaman karmaşıklığına, bellek karmaşıklığından daha fazla önem verilir. Zaman karmaşıklığı ise çoğunlukla algoritma karmaşıklığı olarak alınır. Bir algoritmada toplam adım sayısı $f(n)$ ile sembolize edilsin. $n > n_0$ olmak üzere $f(n) < c \cdot (g(n))$ olmasını sağlayacak bir $c \in R$ sabiti bulunabiliyorsa $f(n) = O(g(n))$ yazılır ve "algoritma karmaşıklığı $g(n)$ derecesindedir" denilir. Örneğin daha önce verdiğimiz algoritmalara ilişkin;

$T_{sum}(n) = 2n + 3 = O(n)$

$T_{rsum}(n) = 2n + 2 = O(n)$

$T_{add}(rows, cols) = 2 \cdot rows \cdot cols + 2 \cdot rows + 1 = O(rows, cols) \Rightarrow n = rows = cols$ ise $T_{add}(n) = O(n^2)$ olur.

Genelde adım sayısı $f(n)$, n'nin bir fonksiyonu olarak düşünülür. Zaman karmaşıklığında (yada algoritma karmaşıklığında) ilgilenilen konu n artarken $f(n)$'in nasıl artacağıdır. Örneğin $f(n) = 3n^3 + 6n^2 + 7n$ olsun.

$3n^3 + 6n^2 + 7n \le 3n^3 + 6n^2 + n^2$ ($n \ge 7$ için)

$3n^3 + 6n^2 + 7n \le 3n^3 + 7n^2$

$3n^3 + 6n^2 + 7n \le 3n^3 + n^3$

$3n^3 + 6n^2 + 7n \le 4n^3 \Rightarrow f(n) = 3n^3 + 6n^2 + 7n = O(n^3)$

Bu sonuç nasıl yorumlanır?

1. İşletim zamanı hiçbir zaman $4n^3$'ü aşamaz.
2. n artarken işletim zamanı kübik olarak artar. Mesela 100 adet tamsayı değerden oluşan bir dizi, söz konusu algoritma ile 2 saniyede sıralanabiliyorsa 300 adet tamsayıdan oluşan dizi en fazla $54 = 2 \cdot 3^3$ saniyede sıralanır. (6 saniye değil)

Algoritma Karmaşıklığı Neden Önemli?

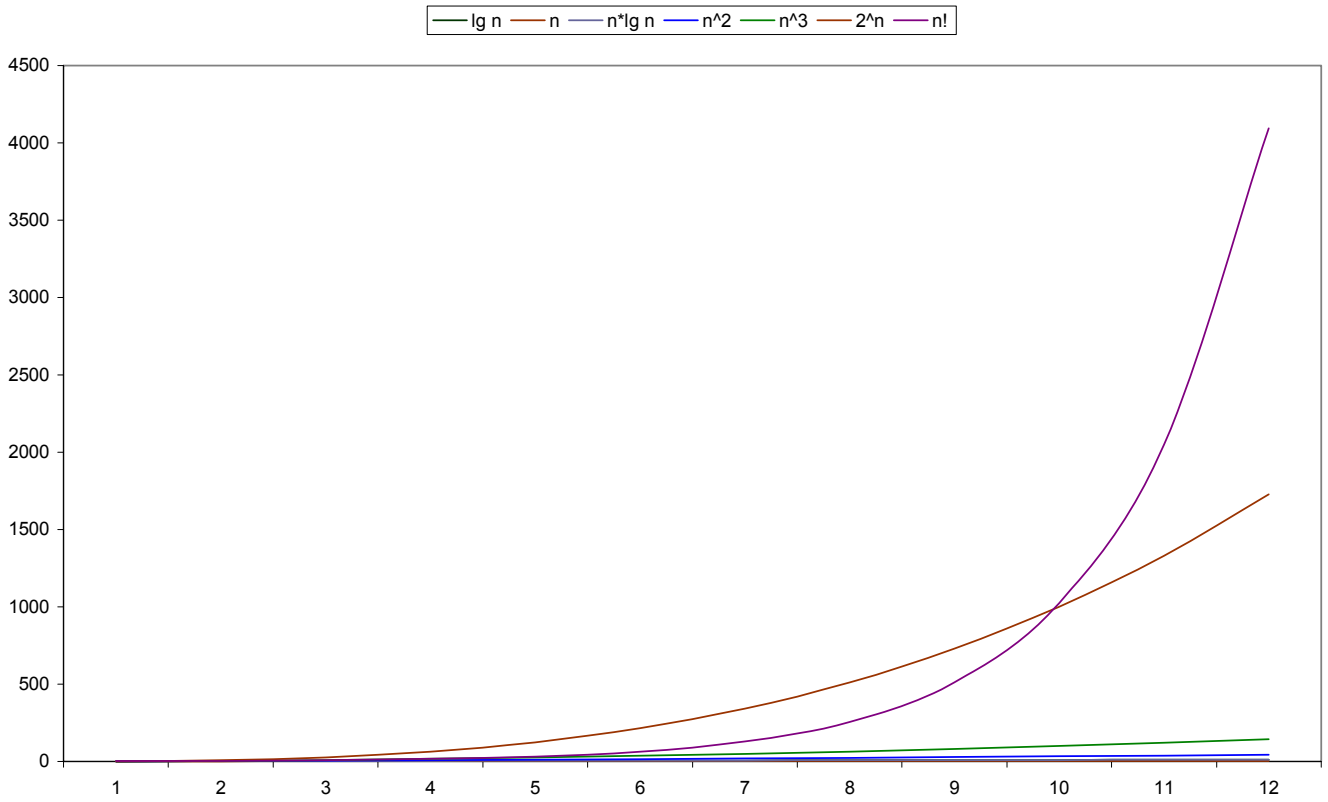
- Algoritmalar arasında hangi alternatifin daha verimli olduğunu bilmek için.
- Veri yapıları arasında tercih yapabilmek için.

ALIŞILMIŞ KARMAŞIKLIKLAR (Practical Complexities) – (s. 37)

Sonsuz sayıda algoritmik karmaşıklık yazılabilir. Bunlar arasında sıkça karşılaşılan bazılarını kıyaslayacağız. Kıyaslamada küçük değerler yanıltıcı olabilir. Belli bir noktadan sonra aşırı artışlar olabileceğine dikkat edilmelidir. 2^n ile n^2 grafiklerini kıyaslayın (s. 39).

Karmaşıklık	Adı	1	2	4	8	16	32
1	**Sabit** (*constant*)	1	1	1	1	1	1
$\lg n$	**Logaritmik** (*logaritmic*)	0	1	2	3	4	5
n	**Doğrusal** (*lineer*)	1	2	4	8	16	32
$n \cdot \lg n$	**Log Lineer** (*loglineer*)	0	2	8	24	64	160
n^2	**Kare** (*quardratic*)	1	4	16	64	256	1024
n^3	**Kübik** (*cubic*)	1	8	64	512	4096	32768
2^n	**Üssel** (*exponential*)	2	4	16	256	65536	4294967296
$n!$	**Faktöriyel** (*factorial*)	1	2	24	40320	$20922789888*10^3$	$263130836933694*10^{21}$

NOT: $\lg n$ ile kast edilen $\log_2 n$'dir.



SİHİRLİ KARELER (Magic Squares) PROBLEMİ – (s. 34)

1'den n^2'ye kadar olan tamsayıların nxn'lik bir kare matrise her satır, sütun ve iki ana köşegen üzerindeki değerlerin toplamı aynı olacak biçimde yerleştirilmesi isteniyor. 5x5'lik kare için toplamlar 65'e eşit olmalıdır.

15	8	1	24	17
16	14	7	5	23
22	20	13	6	4
3	21	19	12	10
9	2	25	18	11

Coxeter Kuralı: İlk satırın ortasına 1 yazılır. Bulunduğumuz karenin sol üstündeki kareye hareket etmeye çalışılır. Bu kare daha önceden işlenmişse bulunduğumuz karenin altındaki kareye hareket edilir. Hareketler sırasında matris dışına taşma varsa en soldaki sütunun en sağdaki sütunun sağında, en alttaki satırın en üstteki satırın üstünde olduğu farz edilir. Bu yöntemle n^2 adet adımda çözüme ulaşırız.

Brute-Force (Kaba kuvvet): Bu kuralı kullanmaz isek $(n^2)!$ farklı olasılığı sırayla tek tek denememiz gerekirdi. Toplam $25! \approx 1.5 \cdot 10^{25}$ olası durumun hepsini tek tek denemek için yaklaşık $4.9 \cdot 10^{16}$ asra tekabül eden bir süre gerekir. (saniyede 1 milyar komut işletilen bir makinede)

```
- Hocam benim Amerika'dan yeni gelen 10GHz'lik bilgisayarım var. Bu problemi veririm,
                hemen çözer. Hiç bunlarla uğraşmama gerek yok.
```

Saniyede 1 milyar komut işletebilen bir bilgisayarda
değişik algoritma karmaşıklıklarına sahip programların gereksinim duyacağı işletim süreleri (s. 40)

n	n	lg n	n*lg n	n^2	n^3	n^4	n^{10}	2^n
10	0.01 µ	0.0033 µ	0.03 µ	0.1 µ	1 µ	10 µ	10 sn	1.024 µ
20	0.02 µ	0.0043 µ	0.09 µ	0.4 µ	8 µ	160 µ	2.8444 saat	1.0486 ms
30	0.03 µ	0.0049 µ	0.15 µ	0.9 µ	27 µ	810 µ	6.8344 gün	1.0737 sn
40	0.04 µ	0.0053 µ	0.21 µ	1.6 µ	64 µ	2.56 ms	4.0454 ay	18.3252 dk
50	0.05 µ	0.0056 µ	0.28 µ	2.5 µ	125 µ	6.25 ms	3.1397 yıl	13.0312 gün
100	0.1 µ	0.0066 µ	0.66 µ	10 µ	1 ms	0.1 sn	32.1502 asır	$4.074*10^{11}$ asır
1000	1 µ	0.01 µ	9.97 µ	1 ms	1 sn	16.67 dk	$3.215*10^{11}$ asır	$3.444*10^{282}$ asır
10000	10 µ	0.0133 µ	132.88 µ	100 ms	16.67 dk	3.86 ay	$3.215*10^{21}$ asır	???

Birim çevirme: **1000 µ = 1 ms** ve **1000 ms = 1 sn**

Performans değerlendirme ikinci yöntemin ***performans ölçümü*** olduğunu ifade etmiştik. Bunun için iki yöntem kullanılabilir. (s. 41)

1- *İşleyici Zamanı (Process Time) (Elapsed Time):* Program başladığında işleyici bu program için kaç saat vuruşu (clock tick) yaptığını hesaplamak üzere otomatik olarak bir sayaç günler.

2- *Sistem Saati (Calender Time):* C programlama dilinin standart kütüphane fonksiyonları kullanılarak işletim zamanı bulunabilir.

	1. YÖNTEM (İŞLEYİCİ ZAMANI)	**2. YÖNTEM (TAKVİM ZAMANI)**
Başlangıç	`Start = clock();`	`start = time(NULL);`
Bitiş	`stop = clock();`	`stop = time(NULL);`
Tür	`Clock_t`	`time_t`
Sonuç (sn)	`duration = ( (double) (stop-start) ) / CLK_TCK`	`duration = ( (double) difftime(stop,start);`

Ölçüm yapılırken dikkat edilmesi gereken bir nokta var. Çevre birimleri ile işlem yapmak fazla vakit alır. Bu, yanıltıcı ölçümlere sebep olabilir. (Çevre birimleri: harddisk, ekran, CD-DVD, yazıcı…) O yüzden, ekrana çıktı gönderen `printf` komutlarını ölçüm yaparken comment etmeliyiz, yani açıklama satırı haline getirmeliyiz.

PERFORMANS ÖLÇÜMÜ UYGULAMALARI

1 – Seçmeli Sıralama (Selection Sort) – (s. 41):

```
#include <stdio.h>
#include <time.h>

#define MAX_SIZE 5001
#define ITERATIONS 20            // sizelist'in eleman sayısı
#define SWAP(x,y,t) ((t)=(x), (x)=(y), (y)=(t))

void sort( int[], int );

int main( void ){
    int i, j, list[MAX_SIZE];    // sıralanacak dizi
    int sizelist[] = {           // farklı n (eleman sayısı) değerleri
        0, 100, 300, 600, 900, 1200, 1500, 2000, 3000, 3100, 3200,
        3300, 3400, 3500, 3600, 3700, 3800, 3900, 4000, 5000};
    clock_t start, stop;
    int duration;

    printf("     Performans Olcumu (Secmeli Siralama)\n"
        "-----------------------------------------------\n");
    for( i=0 ; i<ITERATIONS ; i++ ){
        for( j=0 ; j<sizelist[i] ; j++ )// sizelist[i] adet eleman içeren dizi için
            list[j] = sizelist[i] - j; // dizi elemanları büyükten küçüğe oluşturulur
        start    = clock();
```

```
        sort( list, sizelist[i] ); // sizelist[i] adet eleman içeren dizi sıralanıyor
        stop     = clock();
        duration = (int)(1000 * (stop-start)/CLK_TCK);
        printf("%4d elemani siralama %4d milisaniye surdu\n", sizelist[i], duration);
    }
    return 0;
} // --> end main()

void sort( int list[], int n ){
    int i, j, min, temp;
    for( i=0 ; i<n ; i++ ){
        min = i;
        for( j=0 ; j<n ; j++ )
            if( list[j] > list[min] ){
                min = j;
                SWAP( list[i], list[min], temp );
            }
    }
} // -> end sort()
```

```
     Performans Olcumu (Secmeli Siralama)
-------------------------------------------------
   0 eleman icin siralama    0 milisaniye surdu
 100 eleman icin siralama    0 milisaniye surdu
 300 eleman icin siralama    0 milisaniye surdu
 600 eleman icin siralama    0 milisaniye surdu
 900 eleman icin siralama   15 milisaniye surdu
1200 eleman icin siralama   31 milisaniye surdu
1500 eleman icin siralama   32 milisaniye surdu
2000 eleman icin siralama   62 milisaniye surdu
3000 eleman icin siralama  110 milisaniye surdu
3100 eleman icin siralama   93 milisaniye surdu
3200 eleman icin siralama   94 milisaniye surdu
3300 eleman icin siralama   94 milisaniye surdu
3400 eleman icin siralama  109 milisaniye surdu
3500 eleman icin siralama  125 milisaniye surdu
3600 eleman icin siralama  125 milisaniye surdu
3700 eleman icin siralama  125 milisaniye surdu
3800 eleman icin siralama  141 milisaniye surdu
3900 eleman icin siralama  140 milisaniye surdu
4000 eleman icin siralama  141 milisaniye surdu
5000 eleman icin siralama  234 milisaniye surdu
```

Çıktıda dikkatimizi çeken iki şey var! *Birincisi;* 600 elemana kadarki sıralamalar hiç zaman almamış gözüküyor. *İkinicisi;* 3000 elemanlı dizi 110 ms'de sıralanırken daha fazla elemana sahip diziler 93, 94, 109ms gibi daha az sürelerde sıralanmış gözüküyor. Ayrıca 3500, 3600 ve 3700 elemanlı dizilerin sıralanması eşit vakit almış gibi gözüküyor. Bunlar doğru olamaz! Bu tutarsızlıklar neden kaynaklanıyor? İlkinde sıralama işlemi, iki clock günleme arasında bitmiş, işlem esnasında hiç clock günlenmemiştir. Saat vuruşunun başlangıç ve bitiş değerleri aynı olduğu için `(stop-start)` komutuyla 0 elde ettik. İkincisinde de ilkine benzer bir saat vuruşu tutarsızlığına rastladık. Bu şartlarda aynı eleman sayılı diziler için bile farklı sonuçlar elde etmemiz mümkündü.

Çözüm: Her ölçümü belirli sayıda tekrarlayıp ölçüm sonuçlarının ortalamasını alabiliriz. Bunu sıralı arama programı üzerinde görelim:

2- Sıralı Arama (Sequential Search) – (s. 43):

```
#include <stdio.h>
#include <time.h>

#define MAX_SIZE 5001
#define ITERATIONS 20           // sizelist'in eleman sayısı
inv
int seqsearch( int[], int, int );
```

```c
int main( void ){
    int i, pos, list[MAX_SIZE];  // sıralanacak dizi
    int sizelist[] = {           // farklı n (eleman sayısı) değerleri
        0, 100, 300, 600, 900, 1200, 1500, 2000, 3000, 3100, 3200, 3300, 3400,
        3500, 3600, 3700, 3800, 3900, 4000, 5000};
    long int j, runtimes[] = {    // her bir arama kaç kez tekrar edilecek
        500000, 200000, 100000,    100000, 50000, 50000, 40000, 40000, 30000, 30000,
        25000, 25000, 20000, 20000, 20000, 15000, 10000, 10000, 5000, 5000 };
    clock_t start, stop;
    float duration, total;

    printf("                  Performans Olcumu (Sirali Arama)\n"
        "-------------------------------------------------------------------------\n");
    for( i=0 ; i<MAX_SIZE ; i++ ) list[i] = i;        // dizi elemanlarını doldur
    for( i=0 ; i<ITERATIONS ; i++ ){
        start    = clock();
        for( j=0 ; j<runtimes[i] ; j++ )            // runtimes[i] kez arama yapılır
            pos = seqsearch(list,-1,sizelist[i]);   // dizide -1 aranacak
        stop     = clock();
        total    = (float)(1000*(stop-start)/CLK_TCK);    // toplam arama süresi (sn)
        duration = (float)(1000*total/runtimes[i]); // ortalama bir arama süresi (ms)
        list[sizelist[i]] = sizelist[i];            // son konumdaki -1 düzeltilir

        printf("%4d eleman, %6ld kez arama %3d clock = %4.0f ms. ORT: %6.3f mikro sn\
n",
                sizelist[i], runtimes[i], (int)(stop-start), total, duration);
    }
    return 0;
} // --> end main()

int seqsearch( int list[], int searchnum, int n ){
    int i;
    list[n] = searchnum;    //  aranacak elemanı (-1) son konuma koy
    for( i=0 ; list[i] != searchnum ; i++ );
    return (i<n) ? i : -1;
} // -> end seqsearch()
```

```
                  Performans Olcumu (Sirali Arama)
-------------------------------------------------------------------------
   0 eleman, 500000 kez arama   0 clock =    0 ms. ORT:  0.000 mikro sn
 100 eleman, 200000 kez arama 109 clock =  109 ms. ORT:  0.545 mikro sn
 300 eleman, 100000 kez arama 141 clock =  141 ms. ORT:  1.410 mikro sn
 600 eleman, 100000 kez arama 281 clock =  281 ms. ORT:  2.810 mikro sn
 900 eleman,  50000 kez arama 219 clock =  219 ms. ORT:  4.380 mikro sn
1200 eleman,  50000 kez arama 296 clock =  296 ms. ORT:  5.920 mikro sn
1500 eleman,  40000 kez arama 250 clock =  250 ms. ORT:  6.250 mikro sn
2000 eleman,  40000 kez arama 360 clock =  360 ms. ORT:  9.000 mikro sn
3000 eleman,  30000 kez arama 375 clock =  375 ms. ORT: 12.500 mikro sn
3100 eleman,  30000 kez arama 375 clock =  375 ms. ORT: 12.500 mikro sn
3200 eleman,  25000 kez arama 359 clock =  359 ms. ORT: 14.360 mikro sn
3300 eleman,  25000 kez arama 360 clock =  360 ms. ORT: 14.400 mikro sn
3400 eleman,  20000 kez arama 296 clock =  296 ms. ORT: 14.800 mikro sn
3500 eleman,  20000 kez arama 282 clock =  282 ms. ORT: 14.100 mikro sn
3600 eleman,  20000 kez arama 297 clock =  297 ms. ORT: 14.850 mikro sn
3700 eleman,  15000 kez arama 234 clock =  234 ms. ORT: 15.600 mikro sn
3800 eleman,  10000 kez arama 141 clock =  141 ms. ORT: 14.100 mikro sn
3900 eleman,  10000 kez arama 171 clock =  171 ms. ORT: 17.100 mikro sn
4000 eleman,   5000 kez arama  79 clock =   79 ms. ORT: 15.800 mikro sn
5000 eleman,   5000 kez arama  93 clock =   93 ms. ORT: 18.600 mikro sn
```

Performans ölçümü yaptığımız bilgisayarın clock değeri sabit 1000 olarak kabul edilmiş olduğundan her milisaniyede 1 saat vuruşu oluyor gibi sonuç aldık.

Iterasyonlar, tick sayıları ve saniyeleri içeren tablo var.

BÖLÜM 2 – DİZİLER ve YAPILAR – (s. 49)

ÇOK BOYUTLU DİZİLERİN BELLEKTE GÖSTERİMİ (Representation of Multidimensional arrays) (s. 78)

$A[upper_0][upper_1][upper_2]\dots[upper_{n-1}]$ olarak tanımlanan n boyutlu matrisin $\prod_{i=0}^{n-1} upper_i$ adet elemanı vardır. Ör. $A[10][10][10]$ matrisi 1000 elemanlıdır.

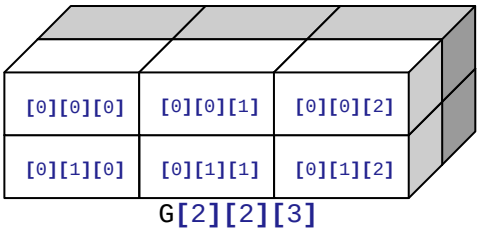
> Eğer performansı artıracaksa makul ölçüde olmak kaydıyla fazladan bellek kullanmakta sakınca yok.

Diziler, belleğe iki şekilde yerleştirilebilir: Satır ya da sütun öncelikli (s. 51).

Satır öncelikli yerleştirme: Satır öncelikli yerleşimde matris satırlara göre yerleştirilir. Örneğin $A[upper_0][upper_1]$ şeklinde verilmiş iki boyutlu bir dizinin $A[0][0]$ elemanının belleğe yerleşim adresi α ise $A[i][0]$ elemanının belleğe yerleşim adresi $\alpha + i \cdot upper_i$ olacaktır. Çünkü $i.$ satırdan önce $i \cdot upper_i$ adet eleman vardır. Dolayısıyla $A[i][j]$ elemanının belleğe yerleşim adresi $\alpha + i \cdot upper_i + j$ olacaktır. `int x[3][4];` şeklindeki dizi:

[0][0]	[0][1]	[0][2]	[0][3]	[1][0]	[1][1]	[1][2]	[1][3]	[2][0]	[2][1]	[2][2]	[2][3]

3 boyutlu bir dizinin satır öncelikli yerleştirilmesi (s. 79): `G[2][2][3]` şeklindeki 3 boyutlu bir dizinin bellek gösterimini 3 boyutlu bir şekille açıklayabiliriz. Açık renkli dilim 0. dilim, koyu renkli dilim 1. dilim olmak üzere bir adres şu şekilde ifade edilir: `G[dilim][satır][sütun]`



G[2][2][3]

$G[upper_0][upper_1][upper_2]$ şeklinde verilen matrisin elemanlarının konumları

eleman	adres
[0][0][0]	α
[i][0][0]	$\alpha + i.upper_1.upper_2$
[i][j][0]	$\alpha + i.upper_1.upper_2 + j.upper_2$
[i][j][k]	$\alpha + i.upper_1.upper_2 + j.upper_2 + k$

$\alpha + i.upper_1.upper_2 + j.upper_2 + k$ şeklindeki son ifade 3 toplama ve 3 çarpma içeriyor. Bir derleyici dizi elemanlarına erişim mekanizmasını hızlandırmak ister. Çarpma, toplamadan çok vakit aldığı için ifadeyi paranteze alarak çarpma sayısını azaltır. İfadeyi paranteze aldığımızda [$\alpha + (i.upper_1 + j).upper_2 + k$] 3 toplama 2 çarpma içeren bir ifade elde etmiş oluruz. Böylece daha hızlı erişim sağlanmış olur.

Bu çalışmamızı $A[upper_0][upper_1][upper_2]\dots[upper_n]$ şeklinde tanımlanmış bir matrisin her hangi bir elemanına erişim için genelleştirelim. Elemanların bellekteki konumları:

$A[0][0]\dots[0]$ α

$A[i_0][0]\dots[0]$ $\alpha + i_0.upper_1.upper_2\dots upper_{n-1}$

$A[i_0][i_1]\dots[0]$ $\alpha + i_0.upper_1.upper_2\dots upper_{n-1} + i_1.upper_2\dots upper_{n-1}$

…

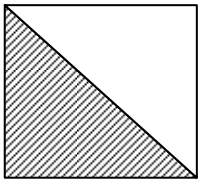
$A[i_0][i_1]\dots[i_{n-1}]$ $\alpha + i_0.upper_1.upper_2\dots upper_{n-1} + i_1.upper_2\dots upper_{n-1} + \dots + i_{n-1}$

$$\&A[i_0][i_1]\dots[i_{n-1}] = \alpha + \sum_{j=0}^{n-1} i_j \cdot a_j \begin{cases} a_j = \prod_{k=j+1}^{n-1} upper_k & 0 \le j < n-1 \\ a_j = 1 & j = n-1 \end{cases}$$ şeklinde formülleştirilebilir.

SİMETRİK MATRİSLER – (s. 93/3)

Bir kare matris köşegenlerinden biriyle bölündüğünde iki parça oluşur. Bu iki parçadan sadece bir tanesini tutmak, kare matrisin programda kullanılma amacını yerine getirmek için bazı durumlarda yeterli olabilir. Örneğin şehirler arası uzaklıkları ya da periyodik tablodaki atom ağırlıklarını tutmak için matrisin yarısı yeterlidir. Böyle bir durumda bellek israfı yapmamak için matrisin diğer yarısı tutulmayabilir. Gerekli veriler tek-boyutlu bir dizide tutulur ve kullanıcıya sanki iki boyutlu bir matris üzerinde çalışıyormuş izlenimi veren bir ara yüz sağlanır. Örneğin;

Kullanıcıya sağlanan arayüz:



$$A = \begin{bmatrix} a_{00} & 0 & 0 & 0 \\ a_{10} & a_{11} & 0 & 0 \\ a_{20} & a_{21} & a_{22} & 0 \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{matrix} \rightarrow 1 \ eleman \\ \rightarrow 2 \ eleman \\ \rightarrow 3 \ eleman \\ \rightarrow 4 \ eleman \end{matrix}$$

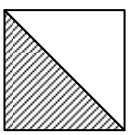
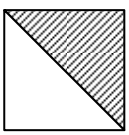
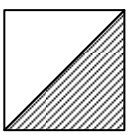
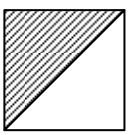
Bellekte gerçek gösterim (ALT dizisi):

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
a_{00}	a_{10}	a_{11}	a_{20}	a_{21}	a_{22}	a_{30}	a_{31}	a_{32}	a_{33}

Görüldüğü gibi (ilk satır numarası 0 olmak üzere) i. satır i+1 adet eleman içermektedir. Buradan hareketle matristeki toplam eleman sayısını söyleyebiliriz: $\frac{n \cdot (n+1)}{2}$ (nxn'lik bir kare matris için) Yani ALT dizisi bu kadar eleman içermelidir. a_{ij} elemanının ALT dizisindeki konumu i. satıra kadarki toplam eleman sayısı ile a_{ij}'nin kendi bulunduğu satırdaki konumunun toplamıdır. Dolayısıyla a_{ij} elemanının ALT dizisindeki konumu: $k = \sum_{t=0}^{i} t + j$ olur.

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_SIZE 100

void AltUcgenOku( int alt[], int n ){   // kullanıcı, diziyi dolduruyor
    int i, j, k;
    if( n*(n+1)/2 > MAX_SIZE ){
        printf("Alt dizi boyutu yetersiz!\n");
        exit(1);
    } else
        for( i=0 ; i<n ; i++ ){
            k = (i+1)*i/2;              // bu satıra kadar kaç eleman yerleşti
            for( j=0 ; j<=i ; j++ )     // satır sütunu geçemez
                scanf("%d", &alt[k+j]); // eleman oku
        }
}

int AltUcgenEris( int i, int j, int n ){
    if( i<0 || i>=n || j<0 || j>=n ){   // Matris boyutları dışında bir
        printf("Gecersiz indis\n");     // ...elemana erisim mi var
        return -2;
    } else if( i < j ) return -1;       // Üst üçgene erişim var (sıfırlı bölge)
    else return (i+1) * i/2 + j;        // Geçerli indis, değeri döndür
}
```

Yazdığımız `AltUcgenEris` yordamının normal dönüş (erişilmek istenen indis) değerleri ile hata değerleri çakışmamalıdır, kesişimleri boş küme olmalıdır. Hata değerleri $\{-1\}$ ve $\{-2\}$ iken normal dönüş değerleri $\{0,1,2,3,...\}$ kümesinin elemanları olur, çakışma yoktur.

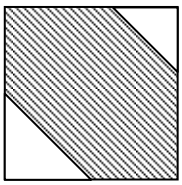





ÇALIŞMA: İlk şekildeki simetrik matris için gerekli mekanizmalar üzerinde çalıştık. Siz de diğer 3 türdeki simetrik matrise erişim mekanizmalarını kurunuz.

KUŞAK MATRİSLER (Band Matrix) - (s. 93/5)

Sıkça karşılaşılan diğer bir matris türü de kuşak matristir. Belirli bölgenin dışındaki elemanlar kullanılmıyordur ya da hepsinin değeri belli bir sabit değere eşittir.

> Saklama problem değil,
> ama hızlı erişebiliyor musun?



$$A=\begin{bmatrix} d_{00} & d_{01} & 5 & 5 \\ d_{10} & d_{11} & d_{12} & 5 \\ d_{20} & d_{21} & d_{22} & d_{23} \\ -5 & d_{31} & d_{32} & d_{33} \end{bmatrix}$$

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
d_{20}	d_{31}	d_{10}	d_{21}	d_{32}	d_{00}	d_{11}	d_{22}	d_{33}	d_{01}	d_{12}	d_{23}

Kuşak matristeki eleman sayısı nedir?

$n=$ Kare matrisin bir kenarının boyutu (örneğimizde 4),
$a=$ Alt üçgendeki kuşak sayısı (ana köşegen dâhil, örneğimizde 3),
$b=$ Üst üçgendeki kuşak sayısı (ana köşegen dâhil, örneğimizde 2) olmak üzere;

Ana köşegende n tane eleman vardır. Ana köşegendeki ve ana köşegen altı kuşaklarda toplam $n+(n-1)+(n-2)+\ldots+(n-(a-1))$ eleman vardır. Ana köşegen üstü kuşaklardaki eleman sayısı ise $(n-1)+(n-2)+\ldots+(n-(b-1))$ olur. Öyleyse matristeki toplam eleman sayısı $n\cdot(a+b-1)-\frac{a\cdot(a-1)}{2}-\frac{b\cdot(b-1)}{2}$

Bir elemana erişmek: Dikkat ettiğimizde $i-j$ değerinin aynı kuşak içinde aynı sabit değere eşit olduğunu görürüz. Bu değeri kuşak numarası olarak kullanacağız. Örneğin d_{20} ve d_{31} elemanlarının bulunduğu kuşak $2-0=3-1=-2$ numaralı kuşaktır. Öyleyse ana kuşak numarası 0 olur. Negatif nolu kuşaklar alt kuşaklar, pozitif nolu kuşaklar üst kuşaklardır. Boş bölgelere denk düşen bir elemana erişilmek istenirse bunu da kuşak numarasından anlayabiliriz. Kuşak numaraları $[1-a,\quad b-1]$ kapalı aralığında olmalıdır.

> Her şey kısıtlı gibi tahlil ederek mekanizma belirliyorum.

Her kuşağın ilk elemanının bellekteki konumunu tutmak üzere de yardımcı bir ARA dizisi kullanacağız. Dolayısıyla ARA dizisi, kuşak sayısı kadar, yani $a+b-1$ adet eleman içermelidir. (Negatif indis değeri olamayacağı için indisler tablodaki gibi -2 yerine 0'dan başlamalıdır. Kuşak no değerine *a-1* ekleyerek bunun üstesinden geliriz.)

ARA dizisi			
[0]	[1]	[2]	[3]
0	**2**	**5**	**9**
[-2]	[-1]	[0]	[1]

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 101

void KusakMatrisKur( int kusak[], int ara[], int n, int a, int b ){
    int i, k, toplamoge;
    if( n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE ){
        printf("Bellek yetersiz!\n"); exit(-1);
    }
    toplamoge = 0;                          // önceki kuşaklardaki toplam öğe sayısı
    for( i = 1-a ; i <= b-1 ; i++ ){        // kuşak kuşak tara
        ara[i+a-1] = toplamoge;             // ara dizisine kuşak başı konumunu yaz
        for( k=0 ; k <= n-abs(i)-1 ; k++ )        // kuşaktaki elemanlar
            scanf("%d", &kusak[ara[i+a-1]+k]);    // kullanıcıdan al
        toplamoge += n - abs(i);            // sonraki kuşak başına hazırlık
}}

int KusakMatrisEris( int i, int j, int n, int a, int b, int ara[] ){
    if( i<0 || j<0 || i>=n || j>=n ){   // İndis taşması var mı?
        printf("Indis tasmasi\n"); return -3;
    } else if( j-i >= 1 ){              // Üst üçgenden bir elemana erişiliyor
        if( j-i < b )                   // Geçerli bir üst kuşakta mı
            return ara[ j-i + a-1 ] + i;// kuşak + kuşaktaki sıra
        else return -1;                 // Üst sıfırlı bölgeye erişim
    } else if( i-j < a )                // Ana köşegende yada alt kuşakta mı
        return ara[ j-i + a-1 ] + j;    // kuşak + kuşaktaki sıra
    else
        return -2;                      // Alt sıfırlı bölgeye erişim
}
```

Yazdığımız `KusakMatrisEris` yordamının karmaşıklığı $O(1)$ olur. Çünkü girdi karakteristiğine bağlı olarak yordamın daha uzun süre çalışması gibi bir şey söz konusu değildir, her durumda aynı süre çalışır. Bu yordamı ara dizisi kullanmadan indisi 0'dan başlayıp döngülerle sayan şekilde yazabilirdik, fakat karmaşıklık artardı. Zaman karmaşıklığı endişesinden ötürü bu şekilde yazmayız. Fakat ARA dizisi kullanmak da hem bellek karmaşıklığını artırır, hem de programı dinamik olmaktan uzaklaştırır. Çözüm olarak ARA dizisini hiç kullanmadan ARA dizisi vazifesi gören bir formül üreterek de kuşak başlangıç değerlerini hesaplayabiliriz.

ÖDEV 1: ARA dizisinden yardım almadan
ÖDEVLERİN ÇÖZÜMLERİ ÖDEVLERİN ÇÖZÜMLERİ…… **DEFTERDE FORLU MORLU BİŞEYLER VAR…**

Kullandığımız derleyici sağdan sola mı tarıyor, soldan sağa mı? Örneğin şu iki eşdeğer ifadeden ikincisi daha güvenlidir:

```
1. if( ifade1 && ifade2 )
```

```
2. if( ifade1)
       if(ifade2 )
```

POLİNOMLAR ÜZERİNDE İŞLEMLER - (s. 59)

Polinomları Tutacak Veri Yapısının Tasarlanması

1. Yöntem (kötü yöntem):

Aşağıdaki gibi bir yapı (`struct`) tanımlarız. `degree`, polinomun en büyük üslü teriminin üssü, yani polinomun derecesidir. `degree` değişkenini kullanmak, her seferinde dereceyi hesaplamaktan bizi kurtarır. `coef` dizisi ise polinomun katsayılarını tutar. (x^i teriminin katsayı `coef[i]`'dedir)

```
#define MAX_DEGREE 101          // Polinomun alabileceği en yüksek derece
typedef struct {
    int degree;                 // En büyük dereceli terimin üssü
    float coef[MAX_DEGREE];     // Katsayılar
} polynomial;
polynomial P;
```

Fakat bu şekildeki gösterimi kullanırsak katsayısı 0 olan terimleri de tutmamız gerekir. Örneğin $P(x)=5x^{100}$ şeklindeki bir polinomu tutmak için 0'dan 99'a kadar 0 değeri içeren bir `coef` dizisi oluşturulacaktır. Bu ise bellek israfıdır.

2. Yöntem (daha iyi yöntem):

Tanımladığımız yapı tek bir terim ($a_i x^i$) tutabilir. Polinomun tamamını tutmak için `polynomial` yapılarının bir dizisini kullanırız. Yazacağımız yordamlarda işimizi kolaylaştırmak için dizideki terimler büyük üslü terimlerden küçük üslülere doğru tutulacaktır. Bu şekilde sıralı tutma, gerektiği yerlerde ***merge mantığı*** kullanabilmemize olanak sağlar.

Merge mantığı, sıralı yapılar üzerinde daha etkin algoritmalar üretmemize olanak sağlar.

```
#define MAX_TERMS 101          // Azami terim sayısı
typedef struct {
    float coef;                // Bir terimin katsayısı
    int expon;                 // Bir terimin üssü
} polynomial;
polynomial P[MAX_TERMS];       // Polinomun tamamı
```

$A(x)=3x^{20}+2x^5+4$			
coef	3	2	4
expon	***20***	***5***	***0***

$B(x)=x^5+10x^3+3x^2+1$				
coef	1	10	3	1
expon	***5***	***3***	***2***	***0***

Polinomların Toplanması – (s. 64)

İki polinomun toplamı: $A(x)+B(x)=\sum(a_i+b_i)\cdot x^i$ şeklinde ifade edilir. Aynı dereceli terimlerin katsayıları toplanır, üsler aynı kalır. Toplama yapmak için iki polinom üzerinde ilerleyen birer gösterge (indis numarası) kullanırız. Terimlerin büyükten küçüğe sıralanmış olması avantajını kullanıp **iki göstergeyle ilerleyerek toplama** yaparız. Şöyle ki;

- A polinomunda sıradaki terimin üssü B'dekiyle aynıysa terimler toplanıp sonuç polinomuna eklenir ve her iki gösterge ilerletilir. (terimler toplandığında katsayı sıfır oluyorsa toplam, sonuç polinomuna eklenmez)
- Eğer A polinomundaki terimin üssü B'dekinden büyükse, *"A'daki üsse sahip bir terim B'de yoktur"* diyebiliriz. Çünkü B'de bir sonraki terimin üssü şuanki terimden daha küçük gelecektir, daha büyük bir terim gelemez. O yüzden A'daki terimi sonuç polinomuna ekler ve göstergeyi ilerletiriz. B'deki göstergeye dokunmayız.
- Benzer şekilde B'dekinin üssü A'dakinden büyük gelirse B'deki terimi ekleyip göstergeyi ilerletiriz.

```
#include <stdlib.h>
#include <stdio.h>
#define MAX_TERMS 101      // Polinomda tutulabilecek azami terim sayısı
#define COMPARE(x,y) ( (x)<(y) ? -1 : (y)<(x) ? 1 : 0 )

typedef struct {
    float coef;            // Bir terimin katsayısı
    int expon;             // Bir terimin üssü
} polynomial;
polynomial P[MAX_TERMS];   // Program boyunca kullanılacak polinom terimleri dizisi
int avail = 0;             // Boş olan ilk konumun indisi

void attach( int coef, int expon ){
    if( avail >= MAX_TERMS ){printf("Dizi tasmasi!\n"); exit(-1);}
    P[avail].expon  = expon;
    P[avail++].coef =  coef;    // avail'i günle
}

void padd(int starta,int finisha,int startb,int finishb,int *startd,int *finishd){
    float coeff;
    *startd = avail;           // sonuç polinomunun başlangıç adresi
    while( starta <= finisha && startb <= finishb )
        switch( COMPARE( P[starta].expon, P[startb].expon ) ){
            // Tek bir polinomda olan elemanı ekle, ilgili polinom göstergesini günle
            case -1: attach( P[startb].coef, P[startb].expon ); startb++; break;
            case  1: attach( P[starta].coef, P[starta].expon ); starta++; break;
            // Her iki polinomda olan elemanı ekle, her iki göstergeyi günle
            case  0: coeff = P[starta].coef + P[startb].coef;
                if( coeff ) attach( coeff, P[starta].expon ); // sıfırsa ekleme
                startb++; starta++;
        }
    // Polinomların birinde kalan elemanlar varsa sonuç polinomuna ekle
    for( ; starta <= finisha ; starta++ ) attach( P[starta].coef, P[starta].expon );
    for( ; startb <= finishb ; startb++ ) attach( P[startb].coef, P[startb].expon );
    *finishd = avail-1;            // sonuç polinomunun bitiş adresi
}
```

Algoritma Karmaşıklığı:

İki sıralı yapıdan bir sıralı yapı elde ederken *merge* mantığını kullandık. Böylece kaydırma ve *insert* etme (araya ekleme) maliyetinden kurtulduk. Karmaşıklığı $O(n^2)$ düzeyinden $O(n+m)$ düzeyine optimize etmiş (iyileştirmiş) olduk. Şöyle ki; A polinomu *n*, B polinomu *m* eleman içeriyor ise en kötü durumda algoritma karmaşıklığı $O(n+m)$ olur. En iyi durum, her iki tarafta da aynı üslü terimlerin bulunduğu durumdur. Bu durumda ise her döngüde case 0'a girileceği için iki gösterge birlikte ilerler ve karmaşıklık $\Omega(n)$ olur.

ÖDEV 2: İki polinomu çarpan `pmult` yordamını yazın ve performans analizini yapın. n tane polinomu nasıl toplardınız? Veri yapısını oluşturun. n tane polinomu nasıl çarpardınız? Veri yapısını oluşturun. (Sayfa 66/5)
ÖDEVLERİN ÇÖZÜMLERİ ÖDEVLERİN ÇÖZÜMLERİ … **DEFTERDE BİRŞEYLER VAR…**

ÖDEV 3: Sayfa 99'daki algoritmayı inceleyin. İlk konumun 2–3 farklı değeri için çıktı oluşturun.
ÖDEVLERİN ÇÖZÜMLERİ ÖDEVLERİN ÇÖZÜMLERİ… **DEFTERDE BİRŞEYLER VAR…**

Polinomların Çarpılması

İki polinomun çarpımı: $A(x) \cdot B(x) = \sum (a_i \cdot x^i \cdot \sum (b_j \cdot x^j))$

A'nın her terimi B'nin her terimi ile çarpılır. Çarpılırken üsler toplanır, katsayılar çarpılır. Oluşan aynı üslü terimler birleştirilir. Farklı terimlerin çarpımıyla oluşacak aynı üslü terimlerin birleştirilmesi ve sıralı yapının korunması gerekmektedir.

```
/* polinomların her bir terimi olarak kullanılacak yapı */
typedef struct pol_oge *pol_gosterge;
typedef struct pol_oge{
    int katsayi;
    int us;
    pol_gosterge bag;
} pol_oge;

/*Listede parametresinden sonra gelen öğeyi listeden ve bellekten siler */
void SonrakiTerimiSil( pol_gosterge Onceki ){
    pol_gosterge Silinecek = Onceki->bag; // free yapabilmek için adresi tutmalıyız
    if( !Silinecek ) return;              // zaten öğe yoksa işlem yapılamaz
    Onceki->bag = Onceki->bag->bag;       // bağ alanları düzgünce güncellenmelidir
    free(Silinecek);                      // bellekteki kalıntı temizlenmelidir
}

/*Dinamik bellek alıp belirtilen üst ve katsayi bilgilerine sahip bir terim oluşturur
  ve Onceki parametresi ile verilen öğenin peşine liste yapısını bozmadan ekler */
pol_gosterge YeniTerimEkle( int Ust, int Katsayi, pol_gosterge Onceki ){
    pol_gosterge p = (pol_gosterge) malloc( sizeof(pol_oge) );
    if( !p ){ printf("Bellek yetmiyor yetmiyoor..."); exit(1); }
    p->us      = Ust;                 // üst     bilgisini yaz
    p->katsayi = Katsayi;             // katsayı bilgisini yaz
    if( Onceki ){                     // Onceki parametresi belirtildi
        p     ->bag = Onceki->bag;    // araya yada sona ekle
        Onceki->bag = p;
    } else p->bag = NULL;     // Onceki boştur, başka bir terimin peşine eklenmeyecek
    return p;
}
/*   Liste olarak verilen iki polinomu birbiriyle çarpıp yeni bir polinom listesi oluşturur ve liste başını döndürür.  */
pol_gosterge PolCarp( pol_gosterge pBas, pol_gosterge qBas ){
    void SonrakiTerimiSil( pol_gosterge Onceki ); // Parametresinden sonra gelen öğeyi listeden ve bellekten siler
    pol_gosterge YeniTerimEkle( int Ust, int Katsayi, pol_gosterge Onceki );/* Yeni bir terim için bellek alıp
                                  üst ve katsayı alanlarını doldurur, Onceki ile verilen öğenin peşine ekler */"
    pol_gosterge baslangic = NULL,          /* yeni oluşturulan bir elemanın sonuç listesi üzerinde
                                               ekleneceği konumun aranmaya başlanacağı yere gösterge */
                 p, q, r,                   // pBas, qBas ve rBas (sonuç) listelerinde dolaşacak göstergeler
                 rBas = NULL;               // yeni oluşturulacak sonuç listesi
    int rUst, rKat;                         // her iki elemanın çarpımında üst ve katsayı bilgileri
    for( p = pBas ; p ; p = p->bag ){       // birinci polinomun her bir terimi için
        r = baslangic;                      // ikinci listeye her başlamada r'yi son işaretlenen konumdan başlat
        for( q = qBas ; q ; q = q->bag ){   // ikinci polinomun her bir terimiyle işlem yapılacak
            rUst = p->us      + q->us;      // üstler toplanır
            rKat = p->katsayi * q->katsayi; // katsayılar çarpılır
            if ( rBas ){                    // listenin ilk elemanı değil
                 // uygun konum ara:
                while  (   r->bag && ( r->bag->us > rUst ) ) r = r->bag;

                // üssü rUst olan eleman daha önce eklenmedi, yeni elemanı ekle:
                if     (  !r->bag || ( r->bag->us < rUst ) ) YeniTerimEkle( rUst, rKat, r );

                // üssü rUst olan eleman zaten eklenmişti, katsayıyı güncelle, katsayı sıfır olursa elemanı sil:
                else if( ( r->bag->katsayi += rKat ) == 0  ) SonrakiTerimiSil(r);
            }
            else rBas = r = baslangic = YeniTerimEkle( rUst, rKat, NULL );   // listenin ilk elemanı eklendi
            if( q == qBas ) baslangic = r;// p listesinin sonraki elemanıyla çarpımlar burdan itibaren yerleşecek
    }}
    return rBas; // sonuç listesinin başını döndür
}
```

Problemin bazı diğer çözümleri:

1. $O(n^2m^2)$: Sıradan hepsini çarp. Her geleni sona ekle. Sonra sırala ver birleştir.
2. $O(n^2m^2)$: Sıradan hepsini çarp. Her seferinde baştan itibaren tarayıp uygun konuma ekle.
3. $O(n^2m)$: İlk polinomun her elemanıyla çarpım için bir liste oluştur ve her adımda sonuçla merge et.
4. $O(\min(n,m)^2 \cdot \max(n,m))$: 3. yöntemin polinomların eleman sayılarına göre optimize edilmiş hali.
5. $O(m \cdot n \cdot \lg(n))$: İlk polinomun her elemanıyla çarpım için ayrı liste oluştur ve en son tümünü merge et.
6. $O(m \cdot n \cdot \lg(\min(m,n)))$: 5. yöntemin polinomların eleman sayısına göre optimize edilmiş hali.
7. Dizi aç, *n.* indis *n.* üssü tutsun. SAKIN HA!

SEYREK MATRİSLER (Sparse Matrix) - (s. 66)

İçerdiği birçok değer sıfır olan matrise seyrek matris denir. Seyrek matrisin kesin tanımını yapmak zordur. Seyrek matris birçok 0 değeri içerdiğinden iki boyutlu bir dizide tutulduğunda bellek kaybına yol açar. Öyleyse alternatif bir matris gerçekleştirimi araştırmalıyız.

Çözüm yollarından biri matrisin her bir elemanını ***<row,col,value>*** üçlüsünden oluşan üçlü (**triple**) bir yapıda tutmaktır. (Başka yöntemler de var. (s. 78)) Normal elemanlar için elemanın satır indisini `row`, sütun indisini `col`, elemanın değerini ise `value` tutar. `M[0]` elemanı diğerlerinden farklıdır, '*başlık*' vazifesi görür. `M[0].row` matristeki satır sayısı, `M[0].col` matristeki sütun sayısı, `M[0].value` ise matriste sıfır olmayan elemanların sayısıdır. Son bir nokta; `M` dizisindeki elemanlar şekildeki gibi önce satıra ve sonra sütuna göre sıralı tutulacaktır.

Bellek karmaşıklığı: Matrisin 3'te 1'i dolu olduğunda aynı bellek maliyeti olur. Daha fazla eleman varsa bu yapı tercih edilmez.

	row	*col*	*value*
$M[0]$	6	6	8
$M[1]$	0	0	15
$M[2]$	0	3	22
$M[3]$	0	5	−15
$M[4]$	1	1	11
$M[5]$	1	2	3
$M[6]$	2	3	−6
$M[7]$	4	0	91
$M[8]$	5	2	28

```
#define MAX_TERM 101

typedef struct {
    int row,    // Satır
        col,    // Sütun
        value;  // Değer
} term;

term M[MAX_TERM];
```

	0	1	2	3	4	5
0	15	0	0	22	0	−15
1	0	11	3	0	0	0
2	0	0	0	−6	0	0
3	0	0	0	0	0	0
4	91	0	0	0	0	0
5	0	0	28	0	0	0

Seyrek Matrisin Devriğini (Transpose) Alma – (s. 69)

$M = \begin{bmatrix} 1 & 3 \\ 0 & 4 \end{bmatrix}$ $M' = \begin{bmatrix} 1 & 0 \\ 3 & 4 \end{bmatrix}$ $M[i][j]$ konumundaki eleman devrik matriste $M'[j][i]$ konumuna yerleşir. Yani kısacası satırları sütun olarak yazıyoruz.

```
// Orjinal M matrisinin devriğini alıp T'yi oluşturur
void transpose( term M[], term T[] ){
    int n, i, j, CurrentT;
    n = M[0].value;          // eleman sayısı (0'dan farklı)
    T[0].row = M[0].col;     // T'nin satır sayısı = M'nin sütun sayısı
    T[0].col = M[0].row;     // M'nin satır sayısı = T'nin sütun sayısı
    T[0].value = n;
    if( n>0 ){               // eleman var mı
        CurrentT = 1;        // devrik matris üzerinde gösterge
        for( i=0 ; i<M[0].col ; i++ )
            for( j=0 ; j<=n ; j++ )
                // M içinde sütunu i olanları bul
                if( M[j].col == i ){
                    T[CurrentT].row = M[j].col;
                    T[CurrentT].col = M[j].row;
                    T[CurrentT].value = M[j].value;
                    CurrentT++;
                }
}}
```

Kaydırma, taşıma kullanıyorsan, veri yapısının hakkını vermiyorsun!

Analiz (s. 70): `transpose` yordamı her bir sütun numarası için `M` matrisini baştan sona tarayarak sütun numarası tutan elemanları devrik matrise ekler. Satır numaraları sıralı gittiğinden dolayı devrik matris de sıralı şekilde oluşacaktır. Fakat algoritmanın karmaşıklığı tam dolu matris için $O(cols^2 \cdot rows)$, seyrek matris içinse $O(cols \cdot elem.say)$ olur.

Bunu daha iyi bir algoritma ile de yapabilirdik (s. 71). Matrisi her sütun için tekrar taramak algoritma karmaşıklığını artırıyor. Eğer elemanların yerleşeceği konumları önceden bilseydik, gelen elemanı gereken yere yerleştirerek ilerlerdik. Örneğin ilk satırda 2 eleman, 2. satırda 3 eleman bulunacaksa ilk satır 1 konumundan başladığı için ikinci satır 3 konumundan başlayacak, 3. satır ise 6. konumdan başlayacaktır. Bu yaklaşımla matrisi bir kez tarayarak satır başlarının yerleşmesi gereken yerleri bulabiliriz. İkinci taramada ise elemanları konumlarına yerleştiririz. Böylece iki tarama ile işlemi tamamlayabiliriz. Aşağıda kodu bulunan `fast_transpose` yordamı için karmaşıklık: $num\_cols + 1 + num\_terms + 1 + num\_cols + num\_terms + 1 \quad = 2 \cdot num\_terms + 2 \cdot num\_cols + 3$ $= O(num\_terms + num\_cols) \quad = O(cols + elem.say.)$

	row	col	value
$M[0]$	6	6	8
$M[1]$	0	0	15
$M[2]$	0	3	22
$M[3]$	0	5	−15
$M[4]$	1	1	11
$M[5]$	1	2	3
$M[6]$	2	3	−6
$M[7]$	4	0	91
$M[8]$	5	2	28

	row	col	value
$T[0]$	6	6	8
$T[1]$	0	0	15
$T[2]$	0	4	91
$T[3]$	1	1	11
$T[4]$	2	1	3
$T[5]$	2	5	28
$T[6]$	3	0	22
$T[7]$	3	2	−6
$T[8]$	5	0	−15

```
void fast_transpose( term M[], term T[] ){
    int i, j, row_terms[MAX_COL], num_cols, num_terms, start_pos[MAX_COL];
    T[0].col   = M[0].row;
    T[0].row   = num_cols  = M[0].col;
    T[0].value = num_terms = M[0].value;
    if( num_terms > 0 ){                  // eleman var mı
        for( i=0 ; i<num_cols ; i++ )     // row_terms'e sıfır doldur
            row_terms[i] = 0;
        for( i=1 ; i<=num_terms ; i++ )   // sütun nosu i olanların sayısını bul
            row_terms[M[i].col]++;
        start_pos[0] = 1;                 // 1. indisten başlanacak
        for( i=1 ; i<num_cols ; i++ )     // sütun başlangıç adreslerini hesapla
            start_pos[i] = start_pos[i-1] + row_terms[i-1];
        // kolon değeri i olan üçlünün T'de yerleşeceği ilk adresi bul
        for( i=1 ; i<=num_terms ; i++ ){
            j = start_pos[M[i].col]++;    // elemanın yerleşmesi gereken konum
            T[j].row   = M[i].col;        // değerleri kopyala
            T[j].col   = M[i].row;
            T[j].value = M[i].value;
        }
    }
}
```

> Programcı, program yazdığı dilin işleç önceliklerini iyi bilmelidir.

`fast_transpose` yordamı tek bir dizi kullanılarak da yazılabilirdi. Bellek karmaşıklığı açısından daha etkili olurdu.

Seyrek Matrisleri Çarpma Algoritması - (s. 73)

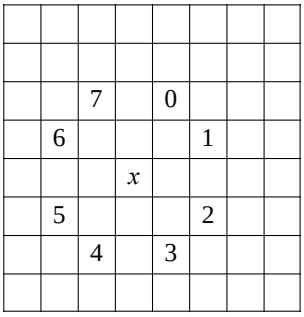
A_{mxn} ve B_{nxp} matrisleri verildiğinde $D_{mxp} = A_{mxn} \cdot B_{nxp}$ çarpım matrisinin bir elemanı $d_{ij} = \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj} \begin{cases} 0 \le i < m \\ 0 \le j < p \end{cases}$ olarak ifade edilir. Pratikte yaptığımız, ilk matrisin satır elemanlarını ikinci matrisin sütun elemanlarıyla çarpıp toplamı sonuç matrisinde ilgili haneye yazmaktan ibarettir. İki seyrek matrisin çarpımı seyrek matris olmak zorunda değildir. Örneğin:

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad \Rightarrow \begin{bmatrix} d_{00} & d_{01} & d_{02} \\ d_{10} & d_{11} & d_{12} \\ d_{20} & d_{21} & d_{22} \end{bmatrix}$$

Bizim kullanmakta olduğumuz üçlü yapı ile tutulan iki matrisi çarpmak için öncelikle ilk matrisin devriğini alacağız. Aksi halde çarpmamız gereken terimleri bulmak için fazladan taramalar yapmamız gerekecektir. 74'te kod var.

SATRANÇ TAHTASINDA AT GEZİSİ – (s. 97/10)

Problemimiz: Bir at, her haneden bir kez geçmek şartıyla satranç tahtasının tüm hanelerinde gezdirilecektir. Atın hareketi, bilindiği üzere, L şeklindedir. Yani 2 yatay 1 dikey ya da 2 dikey 1 yatay konum değiştirebilir.



Çözüm Araştırması: Atı tüm haneleri gezmiş farz edip adım adım geri getirmek suretiyle, yani senaryoyu sondan başa doğru geri sararak problemi çözebiliriz. Bu şekildeki yaklaşıma *back-tracking* (geri sarma) denir. Fakat *back-tracking* bir *brute-force* yöntemidir, karmaşıklığı artırır.

At problemi için 1823 yılında WANSDORFF tarafından *sezgisel bir yaklaşım* geliştirilmiştir.

Sezgisel yaklaşımlar (*heuristic approaches*) geliştirmek için problem üzerinde uzun süre çalışmak ve probleme iyice aşina olmak gerekir. Sezgisel yaklaşımlar problemi 'optimale yakın' çözer, yani akıllıca davranır. Ama tam olarak optimal çözemez. (Mayın tarlası oyununda olduğu gibi)

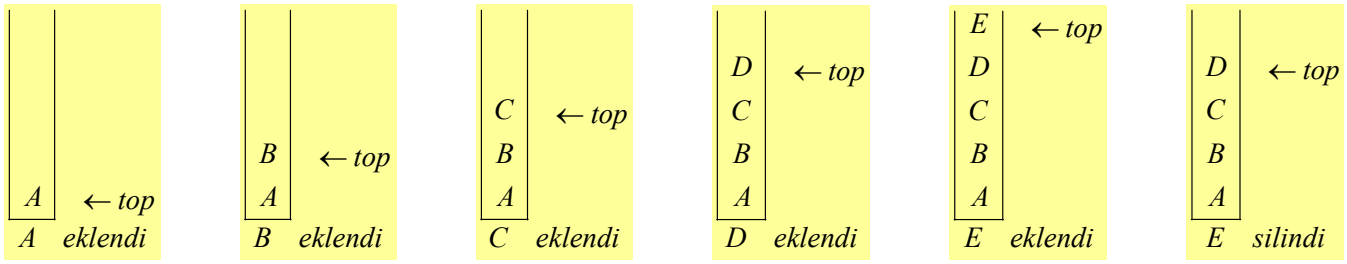
Konum	Hamle	
0	i-2	j+1
1	i-1	j+2
2	i+1	j+2
3	i+2	j+1
4	i+2	j-1
5	i+1	j-2
6	i-1	j-2
7	i-2	j-1

Çözüm: Gelelim WANSDORFF'un sezgisel yaklaşımına… WANSDORFF der ki; "*Atın bir sonraki gideceği yer, ilerleme sayısı sıfır olmayan ama en az olan hanedir.*" İlerleme sayısı demek, o haneden direk olarak hareket edebileceği hanelerin sayısı demektir.

BÖLÜM 2 – YIĞITLAR ve KUYRUKLAR – (s. 101)

YIĞIT ve KUYRUK VERİ YAPISI (Stacks and Queues)

YIĞIT (Stack): Üzerinde ekleme (*insertion*) ve silme (*deletion*) yapılabilen bir listedir. $S = \{a_0, a_1, a_2, \ldots, a_{n-1}\}$ şeklinde bir S yığıtı verildiğinde a_0 alt eleman, a_{n-1} ise üst elemandır. Eğer yığıt veri yapısına sırasıyla A, B, C, D ve E elemanları eklenmişse silinecek, alınacak, işlenecek ilk eleman E olabilir. Son eklenen ilk çıkacağı için **LIFO (last-in-first-out)** mantığına dayalıdır. Tek arabanın sığabileceği bir çıkmaz sokağa park edilen arabalarda da tıpkı yığıt veri yapısında olduğu gibi son giren ilk çıkmak zorundadır. *Stack Activation Frame*, bir sistem yığıtıdır; önceki çerçeveye gösterge, programda nereye dönüleceğini gösteren dönüş adresi ve yerel değişkenleri tutar.

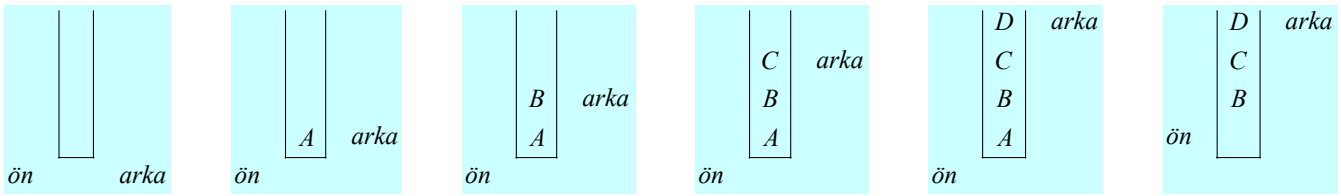


Yığıta Ekleme ve Yığıttan Silme: Yığıtın tepesini `top` değişkeni gösterir. $top < 0$ yığıt boş manasına gelir. Bu haldeyken yığıttan eleman alınmak istenmesi *underflow* (alt taşma) hatasına sebep olur. Yığıt göstergesinin $top \geq MAX\_STACK\_SIZE - 1$ durumunda olması yığıtın dolu olduğu manasına gelir. Bu haldeyken yığıta eleman eklenmek istenmesi *overflow* (üst taşma) hatasına sebep olur.

```c
#include <stdlib.h>
#include <stdio.h>
#define MAX_STACK_SIZE    101

typedef struct {
    int deger;
    // diğer değişkenler
} element;

element yigit[MAX_STACK_SIZE];

void yigit_dolu(){
    printf("Yigit dolu!!!\n"); exit(-1);
}

void yigit_bos(){
    printf("Yigit bos!!!\\n"); exit(-2);
}

void yigit_ekle( int *top, element item ){
    if( *top >= MAX_STACK_SIZE-1 ) // Yeni eleman için yer var mı?
        yigit_dolu();              // Yığıt dolu uyarısı ver ve çık
    yigit[++*top] = item;          // önce göstergeyi artır
    // sonra yığıtta işaret edilen konuma yeni elemanı aktar
}

element yigit_al( int *top ){
    if( *top < 0 )    yigit_bos(); // Yığıt boşsa uyarı ver ve çık
    return yigit[(*top)--];        // Yığıt elemanını döndür, sonra göstergeyi azalt
}
```

Böyle yaparsanız sıfır alırsınız, hatta eksi not alırsınız. Burda bu konuyu kaç kere anlattık değil mi?

KUYRUK (Queue): $Q = \{a_0, a_1, a_2, \ldots, a_{n-1}\}$ kuyruğu verildiğinde yığıt veri yapısından farklı olarak a_0 veri yapısında işlenecek ilk elemandır. a_{n-1} kuyruğun arkasında yer alır. Kuyruğa ilk giren ilk çıkar -> FIFO (first-in-first-out). Sırasıyla A, B, C ve D elemanlarını kuyruğa eklediğimizde ilk alınacak eleman, ilk eklenen eleman olan A'dır.

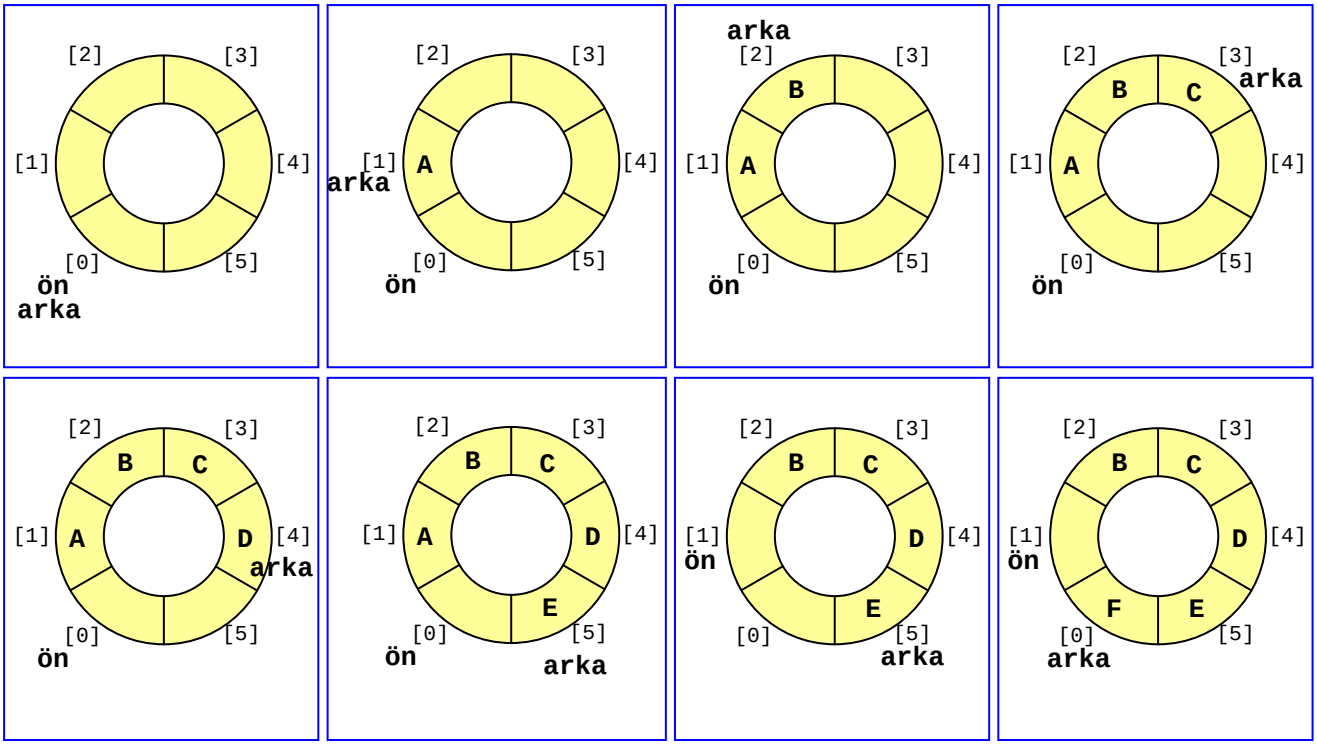



```
#include <stdlib.h>
#include <stdio.h>
#define MAX_QUEUE_SIZE    100

typedef struct {
    int deger;
    // diğer değişkenler
} element;

element kuyruk[MAX_QUEUE_SIZE];
int on   = -1;
int arka = -1;

void kuyruk_dolu(){
    printf("Kuyruk Dolu!!!\n"); exit(-1);
}

void kuyruk_bos(){
    printf("Kuyruk Bos!!!\n"); exit(-2);
}

void kuyruk_ekle( int *arka, element item ){
    if( *arka >= MAX_QUEUE_SIZE-1 )   // Yeni eleman için yer var mı?
        kuyruk_dolu();                // Kuyruk dolu uyarısı ver ve çık
    kuyruk[++*arka] = item;           // önce göstergeyi artır
    // sonra kuyruğun arkasına yeni elemanı aktar
}

element kuyruk_al( int *on, int arka ){
    if( *on == arka ) kuyruk_bos();   // Kuyruk boşsa uyarı ver ve çık
    return kuyruk[++*on];  // Göstergeyi artır ve kuyruğun önündeki elemanı döndür
}
```

Döngüsel Kuyruk (Circular Queue): `arka` göstergesi *MAX_QUEUE_SIZE–1* değerine eşit olduğunda kuyruk dolu uyarısı vermek gerekir. Bu arada kuyruktan birçok eleman alındığı için kuyruğun önünde yeni elemanlar eklenebilecek boş yerler oluşmuş olabilir. Kaydırma yapılarak öndeki boş yerler kullanıma sokulmak üzere arkaya taşınabilir, fakat kaydırmalar aşırı zaman alır. Buna çözüm olarak *döngüsel kuyruk* veri yapısı geliştirilmiştir. Döngüsel kuyrukta kuyruğun başı ile sonu birleştirilmiştir. Önde boşalan yerler arkadaymış gibi otomatik olarak kullanıma sokulur.

> Kaydırma!
> Bilgisayar bilimlerinde
> en kötü kelime!
> Yüksek maliyet demek!

Döngüsel kuyruk *MAX_QUEUE_SIZE* boyutunda açılmışsa tutulabilecek eleman sayısı doğrusal kuyruktakinden bir eksik, yani *MAX_QUEUE_SIZE–1* kadardır. Çünkü ön göstergesinin gösterdiği **bir öğe denetim için boş kalmalıdır**.




```
void kuyruk_ekle( int on, int *arka, element item ){
    *arka = (*arka+1) % MAX_QUEUE_SIZE;   // arka göstergesini ilerlet
    if( on == *arka ) kuyruk_dolu();      // kuyruk doluysa hata ver ve çık
    kuyruk[*arka] = item;                 // boş konuma yeni elemanı aktar
}

element kuyruk_al( int *on, int arka ){
    if( *on == arka ) kuyruk_bos();
      // kuyruk boşsa uyarı ver ve çık
    *on = (*on+1) % MAX_QUEUE_SIZE;
      // ön göstergesini ilerlet
    return kuyruk[*on];
      // öndeki elemanı döndür
}
```

Dünyadaki bilgisayar mühendisliği alanında en iyi üniversite hangisi? Harvard mı? Öyleyse *"Harvard Üniversite'sinde okuyorum da, eğer başaramazsam atılacağım. Daha kötü bir okula gitmek zorunda kalacağım."* diye okumanız lazım.

Labirent (Maze) Problemi – (s. 117)

1	1	1	1	1	1	1	1
1	0	1	0	0	0	1	1
1	0	1	0	1	1	1	1
1	1	0	1	1	1	1	1
1	0	1	1	0	1	1	1
1	1	0	0	1	0	1	1
1	1	1	1	1	1	0	1
1	1	1	1	1	1	1	1

nxp boyutlarında olan bir matriste 0'lar açık yolu, 1'ler ise duvarları simgelemektedir. Giriş konumu `[0][0]`, çıkış konumu `[n][p]` olarak kabul edildiğinde girişten çıkışa hangi yoldan gidilebileceğini bulmak istiyoruz. Yatay ve dikey doğrultuda hareket edebileceğimiz gibi çapraz yönlerde de hareket edebiliriz. Her defasında yasak bir hareket (matris dışına çıkmaya teşebbüs) yapılıp yapılmadığını denetlemekten kurtulmak için matrisin çevresini 1 ile çevirebiliriz. Hareketlerimize yardımcı olması için olası 8 hareket yönünü tutmak üzere bir tablo yapısı tanımlayabiliriz.

yön	hareket[yön].dikey	hareket[yön].yatay
0	-1	0
1	-1	1
2	0	1
3	1	1
4	1	0
5	1	-1
6	0	-1
7	-1	-1

[i-1][j-1]	[i-1][j]	[i-1][j+1]
[i][j-1]	[i][j]	[i][j+1]
[i+1][j-1]	[i+1][j]	[i+1][j+1]

7	0	1
6	**yön**	2
5	4	3

```c
#define MAX_STACK_SIZE  100
#define FALSE           0
#define TRUE            1
#define CIKIS_SUTUN     6
#define CIKIS_SATIR     6

typedef struct { short int dikey, yatay; } yonler;
yonler hareket[8] = {
    {-1, 0}, {-1,  1}, {0,  1}, { 1,  1},
    { 1, 0}, { 1, -1}, {0, -1}, {-1, -1}};

typedef struct { short int satir, sutun, yon; } element;
element yigit[MAX_STACK_SIZE];
int top = -1;

short int  gezildi[8][8];
short int labirent[8][8] = {
    {1,1,1,1,1,1,1,1},
    {1,0,1,0,0,0,1,1},
    {1,0,1,0,1,1,1,1},
    {1,1,0,1,1,1,1,1},
    {1,0,1,1,0,1,1,1},
    {1,1,0,0,1,0,1,1},
    {1,1,1,1,1,1,0,1},
    {1,1,1,1,1,1,1,1}};

void izbul( void ){
    short int satir, sutun, y_satir, y_sutun;
    int i, yon, bulundu = FALSE;
    element konum;
    yigit[0].satir = yigit[0].sutun = yigit[0].yon = 1; // ilk konum bilg. yığıta koy
    gezildi[1][1] = 1;    top = 0;        // ilk konuma hareket edildi
    while( top>-1 && !bulundu ){          // yığıt doldu mu yada çıkış bulundu mu
        konum = yigit_al(&top);           // yığıttan yeni konum bilgilerini al
        satir = konum.satir;              // hangi satır ve sütundan
        sutun = konum.sutun;
        yon   = konum.yon;                // hangi yöne doğru gidilecek
        while( yon<8 && !bulundu ){
            y_satir = satir + hareket[yon].dikey;
            y_sutun = sutun + hareket[yon].yatay;
            if( y_satir == CIKIS_SATIR && y_sutun == CIKIS_SUTUN )
                bulundu = TRUE;           // çıkış noktasında mıyız
            else if( !labirent[y_satir][y_sutun] && !gezildi[y_satir][y_sutun] ){
                gezildi[y_satir][y_sutun] = 1;
                konum.satir = satir;      // bu konuma nereden gelindi
                konum.sutun = sutun;
                konum.yon   = ++yon;      // geri dönersek ilk bakılacak yön hangisi
                yigit_ekle( &top, konum );// yığıta bu bilgileri koy
                satir = y_satir;          // yeni konumu hareket nokt. olarak günle
                sutun = y_sutun;
                yon   = 0;
            }
            else ++yon; // yeni konuma hareket edilemez, diğer yönlere bak
        }    // tüm yönler denendi, geri adım at
    }
    if( bulundu ){
        printf("\nizlenen yol:\nsatir sutun\n");
        for( i=0 ; i<=top ; i++ )
            printf("\n%d %d", yigit[i].satir, yigit[i].sutun );
        printf("\n%d %d", satir, sutun );
        printf("\n%d %d", CIKIS_SATIR, CIKIS_SUTUN );
    } else printf("Cikis yolu bulunamadi!\n");
}
```

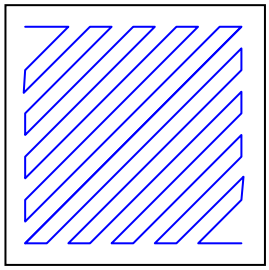
Bana algoritmanızı söyleyin, size…

Değerlendirme:

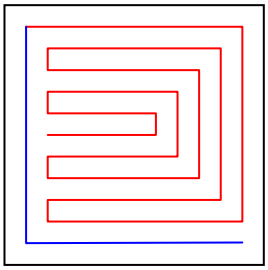
1. Sekizlik *look-up table* kodu kısalttı ve işi kolaylaştırdı.
2. Program, *maze*'i dolaşmak için *stack* yapısını kullanarak *brute-force* yapıyor.
3. En kötü durum analizi için en kötü veri grubunu bulmalıyız. Programın yönleri tarama sırasını dikkate alarak şekildeki iki durumu inceleyelim. Girdi 1'de ilk denenecek olan yol uzun bir yoldur, ama doğru yoldur. Bu yüzden sadece yığıta ekleme sayısı fazla olur, yığıttan silme yapılmaz. Girdi 2'de ise ilk denenecek olan yol uzun, üstelik yanlış yoldur. Bu yolun sonuna kadar gidilir ve çıkmaz olduğu anlaşılır, yığıttaki veriler bir bir silinir. Matrisin doğru yol dışında kalan kısmı tamamen dolaşılarak bunca vakit kaybedildikten sonra doğru yoldan ilerlenip çözüme ulaşılır. İşte bu, bizim programımız için en kötü veri grubudur.

Kendi kodunu ancak sen ayrıntılı inceleyebilirsin. Bir başkası senin kodunu tam olarak analiz edemez.

Sayınız 120-130 değil de 30-40 olsa ve yanımda da 4-5 tane asistanım olsa o zaman ödevlerinizi daha ayrıntılı inceleme imkanı bulabilirdim.



Girdi 1



Girdi 2

Analizini bile beceremediğin algoritmaya burun kıvırma!

İFADE DEĞERLENDİRME (Evaluation of Expressions) – (s. 122)

Mantıksal ve aritmetik ifadelerin derleyiciler tarafından değerlendirilmesi sırasında da yığıt veri yapısı kullanılır. Acaba $x = a/b - c + d \cdot e - a \cdot c$ şeklindeki bir ifadeyi çözümlerken, $((4/2)-2)+(3\cdot 3)-(4\cdot 2)$ gibi bir ifadenin değerini hesaplarken ya da $a/(b-c)+d\cdot(e-a)\cdot c$ gibi parantezli bir ifadeyi işlerken derleyiciler sorunun üstesinden nasıl geliyor?

Problemi önce araştır, o soru başkaları tarafından defalarca çözülmüştür. En hızlı yolunu bul. Çözülmemişse, o zaman sen keşfet.

İfadelerin Farklı Gösterimleri: Alışa geldiğimiz ifadeler *infix (ara-işleç)* biçimindedir. Ara-işleç gösteriminde her işleç, kendi işlenenlerinin arasında yer alır. Bundan başka *prefix (ön-işleç)* ve *post-fix (son-işleç)* biçiminde olmak üzere iki gösterim daha vardır. Ara-işleç gösteriminde hem parantezleri, hem de işlem önceliklerini dikkate almamız gerekir. Oysa ön-işleç ve son-işleç gösterimlerinde işlem önceliğini düşünmeden işlem yapabiliriz.

Örnek No:	Ön-işleç (pre-fix)	Ara-işleç (in-fix)	Son-işleç (post-fix)
1	x34	3x4	34x
2	2+x34	2+3x4	234x+
3	−x+abc/da	(a+b)xc − d/a	ab+cxda/−
4	xx/a+−bcd−eac	((a/(b−c+d))x(e−a)xc	abc−d+/ea−xcx

Son-İşleci Ara-İşlece Çevirme (Postfix to Infix)

Son işleç biçimindeki 42/2−33x+42x− ifadesini ara işleç biçimine adım adım çevirelim:

	YIĞIT			
token	[0]	[1]	[2]	top
4	4			0
2	4	2		1
/	(4/2)			0
2	(4/2)	2		1
-	((4/2)-2)			0
3	((4/2)-2)	3		1
3	((4/2)-2)	3	3	2
x	((4/2)-2)			1
+	(((4/2)-2)+ (3x3))			0
4	(((4/2)-2)+ (3x3))	4		1
2	(((4/2)-2)+ (3x3))	4	2	2
x	(((4/2)-2)+ (3x3))			1
-	((((4/2)-2)+ (3x3))-(4x2))			0

Son-İşleç Biçimdeki İfadeyi Hesaplama (Postfix Evaluation)

Son işleç biçimindeki ifadenin `'\0'` ile sonlandığını ve işlenenlerin birer karakterden (basamaktan) oluştuğunu varsayalım. İfadeyi karakter karakter okuyup okunan her bir token (damga - söz dizim birimi) için bir işlem yapacağız. Karakter olarak okuduğumuz rakamları sayıya çevirmek için `token-'0'` komutunu kullanabiliriz. Bu komut `token` karakterinin ASCII kodundan 0 rakamının ASCII kodu olan 48'i çıkarır, böylece sayı değeri elde edilmiş olur. Örneğin `token` karakteri ASCII kodu 50 olan 2 sayısı ise 50-48=2 işlemi ile 2'nin sayı değeri elde edilir.

```
#define MAX_SIZE    100

typedef enum {
    sol_paran, sag_paran, arti, eksi,
    carpma, bolme, mod, eos, operand
} oncelik;

float yigit[MAX_SIZE];
char expr[] = "42/2-33*+42*-\0";

oncelik token_al( char *sembol, int *n ){
    *sembol = expr[(*n)++];
    switch( *sembol ){
        case '(': return sol_paran;
        case ')': return sag_paran;
        case '+': return arti;
        case '-': return eksi;
        case '/': return bolme;
        case '*': return carpma;
        case '%': return mod;
        case '\0':return eos;
        default:  return operand;
    }
}
```

> **`typedef enum { ... };`**
> ile etiketleme yapmak işimizi kolaylaştırır.

> Fonksiyonları component (bileşen) yazar gibi yaz. Global değişken, define gibi kullanımlarla dışa bağımlılık oluşturma!
> Böyle component (bileşen) yazılmaz! Yarın bir yerde işe girip ekip halinde çalıştığın zaman proje yöneticisine şöyle mi diyeceksin: Ben çok has programcıyım. Kendime özgü bir tarzım var. Benim değişkenleri main'in üstüne koy. Ben böyle çalışıyorum. Beni kabul edersen olduğum gibi kabul et, yoksa...

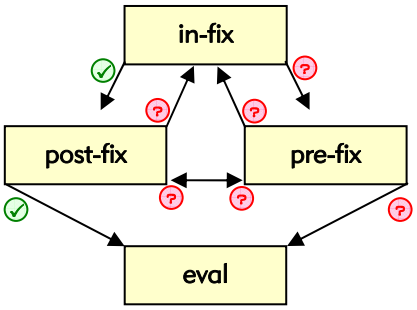
```
float eval(){
    // son işleç biçiminde verilen aritmetik ifadeyi değerlendirip sonucu döndürür
    char sembol;
    oncelik token;
    float op1, op2;
    int n = 0, top = -1;
    token = token_al( &sembol, &n );      // bir söz dizim birimi (token) al
    while( token != eos ){                // karakter kümesi sonu (eos) mu
        if( token == operand )            // token işlenen (operand) mi
            yigit_ekle( &top, sembol-'0' );    // yığıta koy
        else {                            // bir işleç (operatör) geldi
            op2 = yigit_al( &top );       // yığıttan iki
            op1 = yigit_al( &top );       // ... işlenen al
            switch( token ){              // işlenenleri işle ve yığıta koy
                case arti:      yigit_ekle( &top, op1 + op2 ); break;
                case eksi:      yigit_ekle( &top, op1 - op2 ); break;
                case carpma:    yigit_ekle( &top, op1 * op2 ); break;
                case bolme:     yigit_ekle( &top, op1 / op2 ); break;
                case mod:       yigit_ekle( &top, (int)op1 % (int)op2 );
            } // -> end switch
        } // -> end if-else
        token = token_al( &sembol, &n ); // bir token al
    } // -> end while
    return yigit_al( &top );    // karakter kümesinin sonuna gelindi
    // yığıtta kalan tek eleman sonuç olmalıdır.
}
```

Ara-İşleci Son-İşlece Çevirme (Infix to Postfix)

infix: **a+b*c**			postfix: **abc*+**	
token	**[0]**	**[1]**	**top**	**Çıktı**
a			-1	a
+	+		0	
b	+		0	ab
*	+	*	1	ab
c	+	*	1	abc
eos			-1	abc*+

infix: **a*(b+c)*d**				postfix: **abc+*d***	
token	**[0]**	**[1]**	**[2]**	**top**	**Çıktı**
a				-1	a
*	*			0	a
(	*	(		1	a
b	*	(		1	ab
+	*	(	+	2	ab
c	*	(	+	2	abc
)	*			0	abc+
*	*			0	abc+*
d	*			0	abc+*d
eos					abc+*d*

infix_to_postfix kodu gelecek ama son işleneni basamıyor???



ÇALIŞMA: İfade çözümleme ile ilgili diğer çevirme ve hesaplama algoritmalarını tasarlayınız.

ÇOKLU YIĞIT ve KUYRUKLAR (Multiple Stacks and Queues)

Tek boyutlu bir dizi (memory[MEMORY_SIZE]) n adet yığıt kurmak için kullanılabilir. Bu durumda diziyi n adet kesime (segment) bölmemiz gerekir. Dizi boyutu m=MEMORY_SIZE olarak kabul edildiğinde ve yığıtların top ve sınır değerlerini tutacak iki yardımcı dizi kullanıldığında veri yapısı şu şekilde olur:

LİSTELER (Lists)

En güçlü veri yapısı dizidir, ama her şey diziyle çözülemiyor. Örneğin diziye ekleme ve diziden silme işlemleri, tek bir eleman için $O(n)$'lik bir kaydırma maliyeti getiriyor. Bu durumda veri yapısı olarak dizi tercih edilmez, liste tercih edilir. Çünkü listeler için fiziksel olarak kaydırma yapmıyoruz, sadece tek bir öğenin *'sonraki öğe'* bilgisini güncelliyoruz.

```
Pointerları bu yaşlarda
öğrendin öğrendin, yoksa 4.
sınıfta bile pointer'ı
öğrenmemiş olursun.
```

Bellek adresleri iki parçadan oluşur: Kesim adresi, kesim içi adresi. ……..

```
- Ben iki derstir ne anlatıyorum? Dersi bıraktım, C anlattım, pointerları anlattım. Çok
iyi bildiğiniz pointerları bir de ben anlattım.
- Ben pointerları çok iyi biliyorum, ama hoca bir şey diyor, yine de anlamadım.
- Olabilir.
```

Normal şartlarda yordamın formal parametre listesinde "*parametre" olarak gözüken değişken, yordam içinde de "*parametre" olarak kullanılmalıdır.

Her pointer'a mallocla yer ayırmak gerekmiyor, yok öyle bir şey!

List göstergesiyle işaret edilen sıralı listenin bağ alanlarını tersine çeviren `liste_devrik_al` yordamını yazınız. Oge_ekle, oge_cikar, malloc, ekstra bellek, eksta işler yok; hiçbir şey yok!

```
typedef struct liste *list_gosterge ;
typedef struct liste {
    int deger;
    struct liste *bag;
};

void liste_devrik_al ( list_gosterge *list ){
    list_gosterge  r,            // en gerideki gösterge
                   p = *list,    // en ilerideki gösterge
                   q = NULL;    // ortadaki gösterge
    while( p ){
        r = q;         // r, q'yu takip eder.
        q = p;         // q, p'yi takip eder.
        p = p->bag;     // p'yi ilerlet.
        q->bag = r;     // daha önce q, r'den önce yer alıyordu, tersine çevir.
    }
    *list = q;         // son öğe liste başı.
}
```

Bağlaçlı Liste Kullanarak Yığıt Gerçekleştirimi - (s. 148)

Top, liste başını tuttu. Silerken son elemandan geri gelmek gerektiği için ya iki işaretçi ile sonu tut, ya da tek işaretçi ile başı tut.

Seyrek Matrislerin Döngülü Bağlı Listeyle Gösterimi

> Senin ödev kâğıdını şöyle benim kâğıtla üst üste getirip ışığa tuttuğum zaman… (fark görmemem lazım)