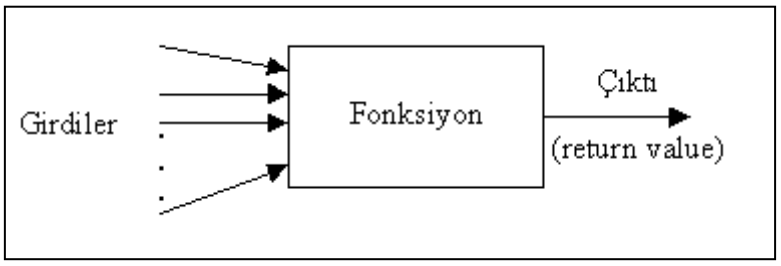
İçindekiler

1. FONKSİYONLAR

Fonksiyon, belirli sayıda verileri kullanarak bunları işleyen ve bir sonuç üreten komut grubudur. Her fonksiyonun bir adı ve fonksiyona gelen değerleri gösteren parametreleri (bağımsız değişkenleri) vardır. Genel olarak bir fonksiyon Şekil'deki gibi bir kutu ile temsil edilir:



Fonksiyonların girdilerine *parametre* denir. Bir fonksiyon bu parametreleri alıp bir işleme tabi tutar ve bir değer hesaplar. Bu değer, *çıktı* veya *geri dönüş değeri* (return value) olarak adlandırılır. Unutmayın ki, bir fonksiyonun kaç girişi olursa olsun sadece bir çıkışı vardır. [1]

C Programlama Dili, kullanıcısına bu türden fonksiyon yazmasına izin verir. C dilinde hazırlanan bir fonksiyonun genel yapısı şöyledir:

```
FonksiyonTipi FonksiyonAdı(parametre listesi)
parametrelerin tip bildirimleri
{
  Yerel değişkenlerin bildirimi
  ...
  fonksiyon içindeki deyimler veya diğer fonksiyonlar
  ...
  return geri dönüş değeri;
}
```

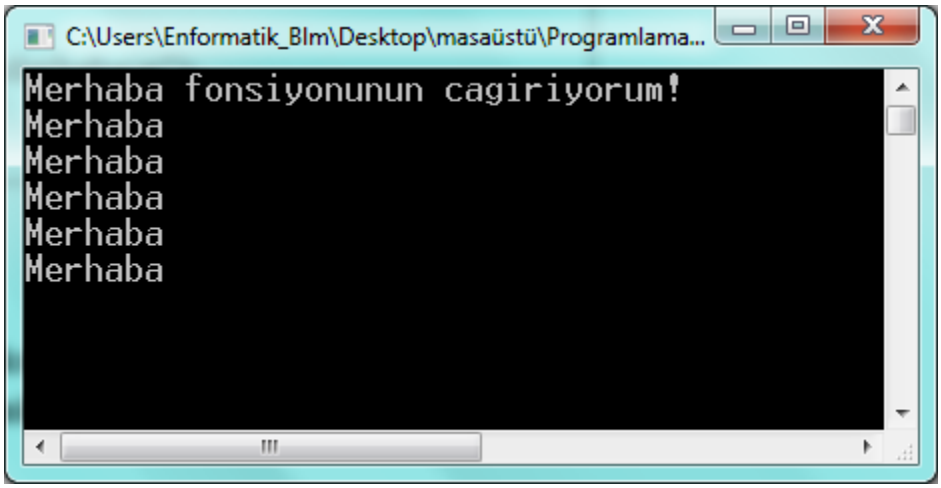
Örneğin iki sayının toplamının hesaplayacak bir fonksiyon şöyle tanımlanabilir:

```
int topla(int x, int y)
{
  int sonuc;
  sonuc = x + y;
  return sonuc;
}
ya da
int topla(int x, int y)
{
  return (x+y);
}
```

1.1. Parametreli Fonksiyonlar

Parametreli fonksiyonlar belirli bir veya birkaç giriş değerine göre işlem yapan fonksiyonlardır. Bu fonksiyonlar çağırıldıkları yere değer döndürebilmekle beraber değer döndürmek gibi bir zorunlulukları da yoktur. Parametre olarak fonksiyonlara değişken yada işaretçi tanımlanabilir. Aşağıdaki örnekte tamsayı tipinde parametre alan ve bu parametre kadar ekrana merhaba yazdıran geriye bilgi döndürmeyen (void) merhaba isimli bir fonksiyon görülmektedir.

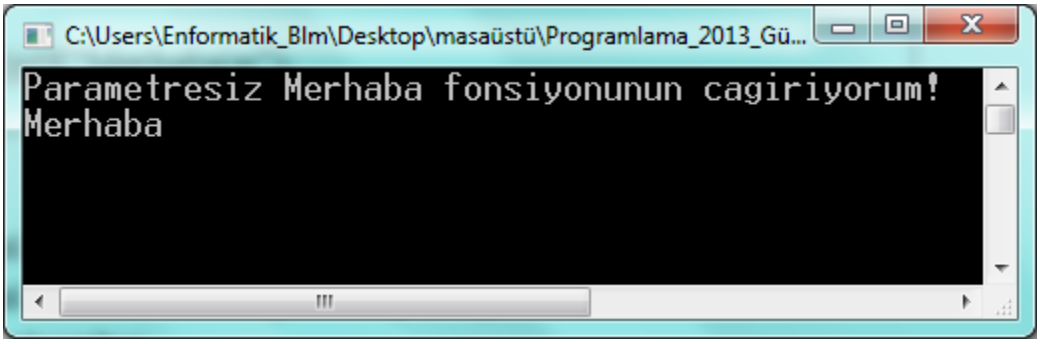
```
#include<stdio.h>
#include<conio.h>

void merhaba(int x);//prototype tanımlama

main()
{
	int a = 5;

	printf("Merhaba fonsiyonunun cagiriyorum!\n");
	merhaba(a); // a değişkeni kadar ekrana merhaba
	getch();
}

void merhaba(int x)
{
	for (int i=0;i<x;i++)
	{
		printf("Merhaba\n");
	}
}
```



1.2. Parametresiz Fonksiyonlar

Parametresiz fonksiyonlar belirli bir giriş değerine göre işlem yapmayan fonksiyonlardır. Bu fonksiyonlar programcı tarafından çağırıldıkları noktada fonksiyon içerisinde tanımlanan görevini yerine getirirler. Aşağıda görevi ekrana sadece bir kez "Merhaba" yazmak olan parametresi olmayan ve çağırıldığı yere çalıştıktan sonra değer döndürmeyen bir fonksiyon bir fonksiyon görülmektedir.

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

void merhaba();//prototype tanımlama

main()
{
    int a = 5;

    printf("Parametresiz Merhaba fonsiyonunun cagiriyorum!\n");
    merhaba();
    getch();
}

void merhaba()
{
            printf("Merhaba\n");
}
```

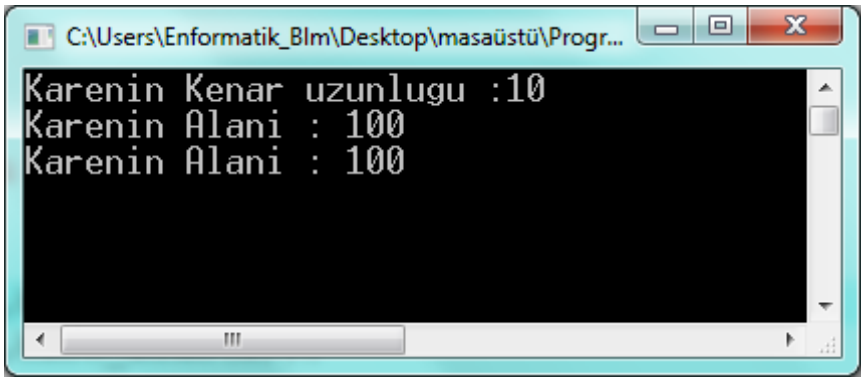


1.3. Geri Dönüş Tipi Belirli Fonksiyonlar

Bu fonksiyon tipi için programcı parametre tanımlayarak ya da parametre tanımlamaksızın fonksiyonu ana (main) fonksiyon içinde çağırdığı noktada fonksiyonun bir değer döndürmesini ister. Yani bu fonksiyon çalıştığında bir değer üretir ve bu değer çağırıldığı yerde bir değişkene aktarılarak ya da ekran çıktısı oluşturacak şekilde kullanılır. Tanımlanan fonksiyonun değer döndürebilmesi için return ifadesinin yer alması gerekir. Konunun daha iyi anlaşılması için örnek bir senaryo oluşturalım.

Örnek senaryoda programcı bir kare için cevre hesaplayan bir fonksiyon tanımlamak istesin. Bu fonksiyon tamsayı olarak gönderilen kenar uzunluk değerine göre karenin alanını hesaplasın ve geriye çağırıldığı yere tamsayı tipinde döndürsün.

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

int kare_alan(int x);//prototype tanımlama

main()
{
	int a;
	int karenin_alani;

	printf("Karenin Kenar uzunlugu :");
	scanf("%d",&a);
	karenin_alani = kare_alan(a);
	printf("Karenin Alani : %d\n",karenin_alani);
	printf("Karenin Alani : %d\n",kare_alan(a));

	getch();
}
int kare_alan(int x)
{
		return (x*x);
}
```



1.4. Geri Dönüş Tipi Belirsiz Fonksiyonlar (Void)

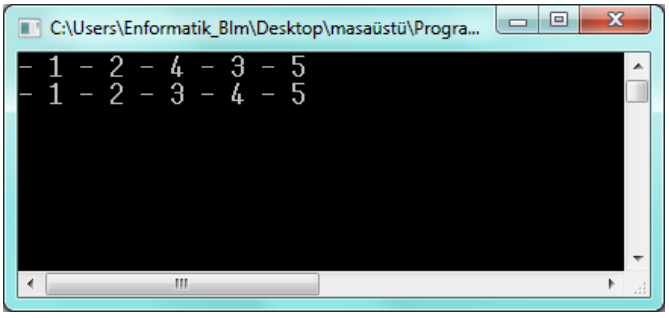
Geri dönüş tipi belirsiz fonksiyonlar programcı tarafından tanımlana parametre kullanan ya da parametresiz olarak tanımlanan çağırıldıkları yere herhangi bir değer döndürülmesi istenmeyen fonksiyon tipleridir. Void fonksiyon olarak ifade edilirler. Aşağıda parametreli iki adet fonksiyon tanımlanacaktır.

Bu fonksiyonlardan birincisi bir dizinin elamanlarını ekrana yazan *dizi_yaz()* fonksiyonu ve dizinin belirlenen iki elemanını yer değiştiren *swap()* fonksiyonlarıdır.

dizi_yaz() fonksiyonu çağırıldığı yerde parametre olarak aldığı dizinin elamanlarını ekrana yazmaktadır.

swap() fonksiyonu çağırıldığı noktada dizinin belirlenen iki elamanını yer değiştirecektir. Bu yer değiştirme işlemi sonrasında kullanıcıya ya da programda çağırıldığı noktaya bir değer döndürmesine gerek yoktur. Görevi sadece iki dizi elemanın yer değiştirmesi olacaktır.

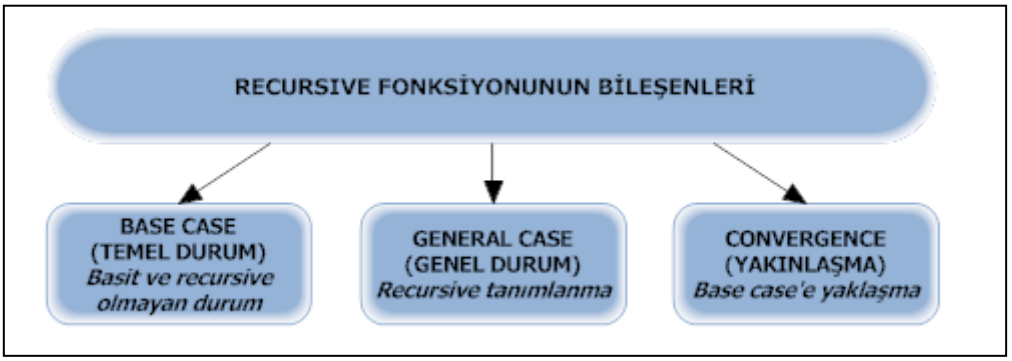
```
#include<stdio.h>
#include<conio.h>

void swap(int x[5], int y);//prototype tanımlama
void dizi_yaz(int k[5]);//prototype tanımlama

main()
{
	int a[5] = {1,2,4,3,5};
	dizi_yaz(a);
	swap(a,3);
	dizi_yaz(a);
	getch();
}
void swap(int x[], int y)
{
	int yedek = x[y];
	x[y] = x[y-1];
	x[y-1] = yedek;
}
void dizi_yaz(int k[])
{
	for(int i=0;i<5;i++)
	{
		printf("- %d ",k[i]);
	}
	printf("\n");
}
```



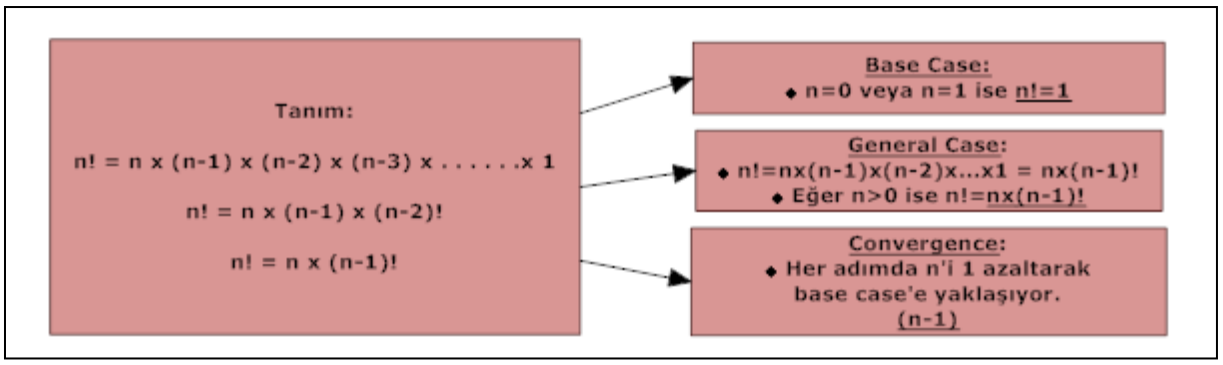
1.5. Kendini Çağırabilen (Recursive) Fonksiyonlar

Özyinelemeli yapı yani kendi kendini çağırabilen (recursion) programlama mantığı temel olarak dögüsel programlama yapılarına alternatif olarak ortaya çıkmıştır. Temelde parametreli bir fonksiyonun belirli bir kısıt değere ulaşana kadar (if kontrol yapısı ile bir değerin kontrolü yapılarak) kendisini başka bir değer ile çağırması olarak ifade edilebilir. [2]

Bir problem özyinelemeli olarak çözülmek istendiğinde aşağıdaki gibi parçalara ayrılmalıdır.



Faktoriyel alma işleminin özyinelemeli olarak hesaplanmasında;



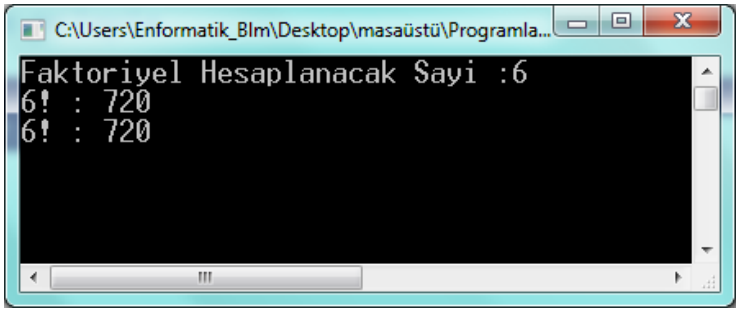
Burada temel durum (Base Case) 1 değerinin ne zaman ya da hangi sayı için faktöriyel alındığında bulunacağının belirlenmesidir. Yani aslında bitiş noktasının tayin edilmesidir.

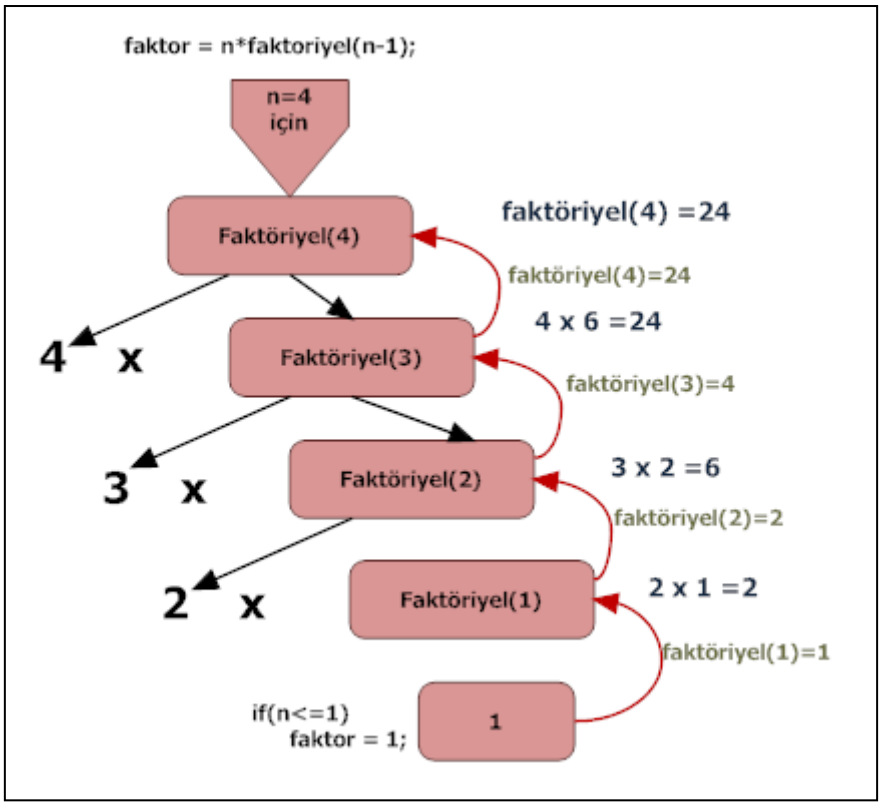
Genel durumda ise problemin çözümüne ait yaklaşım öne sürülür. Örnek olarak n sayısının faktöriyeli aslında n* (n-1)! şeklindedir. n-1 sayısının faktöriyeli de yine (n-1)* ((n-1)-1)! Şeklinde olacaktır.

Yakınlaşma bölümünde ise artırım yada azatlım ifadesi tanımlanmaktadır. Yani base case'a nasıl yaklaşılacaktır bu tanımlanır.

Aşağıda faktöriyel örneği görülmektedir.

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

int rec_fakt(int x);//prototype tanımlama
int fakt(int x);//prototype tanımlama

main()
{
	int sayi;
	int fk_sayi;
	printf("Faktoriyel Hesaplanacak Sayi :");
	scanf("%d",&sayi);

	fk_sayi = rec_fakt(sayi);//Recursive yoluyla
	printf("%d! : %d\n",sayi,fk_sayi);

	fk_sayi = fakt(sayi);//Döngüsel mantıkla
	printf("%d! : %d\n",sayi,fk_sayi);

	getch();
}
int rec_fakt(int x)
{
	int sonuc;
	if(x<=1)
	{
	   sonuc = 1;
	}
	else
	{
		sonuc = x * rec_fakt(x-1);
	}
	return sonuc;
}
int fakt(int x)
{
	int sonuc = 1;
	for(int i=x;i>=1;i--)
	{
		sonuc = sonuc * i;
	}
	return sonuc;
}
```





ÖRNEK: Fibonacci sayıları

Finobacci serisi : 0, 1, 1, 2, 3, 5, 8, 13, 21, …

Tanım : 0 ve 1 ile başlar, sonraki sayılar kendinden önceki iki sayının toplamı ile oluşturulur.

Base Case : fibonacci(0) = 0 fibonacci(1) = 1

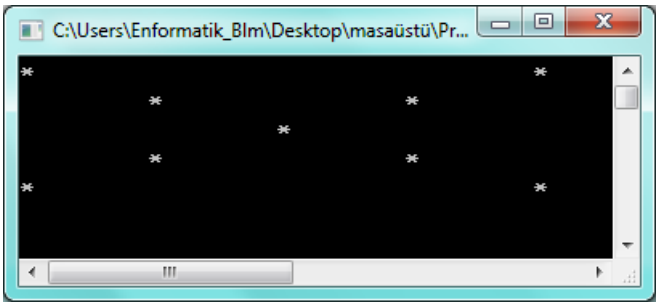
General Case : fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)

Convergence : n azaltılarak base case'e yaklaşılır.

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

int fibonacci(int); /*fonksiyon prototipleri*/

main() {
   int sayi;
   long sonuc;
   printf("Fibonacci degeri bulunacak bir sayi giriniz : ");
   scanf("%d", &sayi);
   sonuc = fibonacci(sayi);
   printf("Fibonacci(%d) = %d", sayi, sonuc);
      getch();
}

int fibonacci(int n) /*Fonksiyon tanimlari*/ {
   int fib;
   if (n <= 1)
      fib = n;
   else
      fib = fibonacci(n - 1) + fibonacci(n - 2);
   return fib;
}
```

1.6. Bölüm Çalışma Soruları

1. Parametre olarak kullanıcı tarafından klavyeden girilen bir tamsayıyı alan ve bu sayı kadar ekrana yatay ve dikey düzlemde (*) işareti koyan bir ***yildiz()*** fonksiyonu tanımlayarak, ana fonksiyon içinde çağırınız.
2. Ekrana aşağıdaki ekran çıktısını verecek parametresiz void bir fonksiyon tanımlayarak ana fonksiyon içinde çağırınız.



3. Parametre olarak kullanıcı tarafından girilen iki adet tam sayıyı alan ve bu tam sayıların ortak bölenlerinin en büyüğünü hesaplayan ve çağırıldığı yere bu değeri döndüren bir ebob() fonksiyonu tanımlayınız ve ana program içerisinde çağırınız.
4. Parametre olarak kullanıcı tarafından girilen bir isim içerisindeki sesli harf sayısını bulan ve ekrana yazan fonksiyonu tanımlayarak ana program içinde çağırınız.
5. Parametre ve b gibi iki adet tamsayı alan ve a üzeri b üs alma işlemini öz yinelemeli olarak yapan bir us_al() fonksiyonu tanımlayarak ana program içinde çağırınız.
6. Parametre olarak iki boyutlu 4x4 bir dizi alan ve bu dizinin bir x eksenine göre aksini (simetriğini) ekrana yazan bir fonksiyonu tanımlayarak ana program içinde çağırınız.

2. İŞARETÇİLER

2.1. Gösterici Nedir?

Bir gösterici, diğer değişkenler gibi, sayısal bir değişkendir. Bu sebeple kullanılmadan önce program içinde bildirilmelidir. Gösterici tipindeki değişkenler şöyle tanımlanır:

```
tip_adı *gösterici_adı;
```

Burada *tip_adı* herhangi bir C tip adı olabilir. Değişkenin önünedeki * karakteri yönlendirme (indirection) operatörü olarak adlandırılır ve bu değişkenin veri değil bir adres bilgisi tutacağını işaret eder. Örneğin:

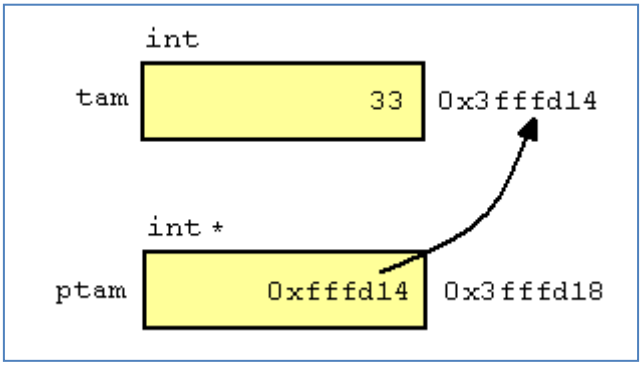
```
char  *kr;          /* tek bir karakter için */
int    *x;          /* bir tamsayı için */
```

Yukarıda bildirilen göstericilerden; `kr` bir karakterin, `x` bir tamsayının ve `deger` bir gerçel sayının bellekte saklı olduğu yerlerin **adreslerini** tutar.

Bir göstericiye, bir değişkenin adresini atamak için adres operatörünü kullanabiliriz. Örneğin tamsayı tipindeki `tam` adlı bir değişken ve `ptam` bir gösterici olsun. Derleyicide, aşağıdaki gibi bir atama yapıldığında:

```
int *ptam, tam = 33;
.
.
.
ptam = &tam;
```

`ptam` göstericisinin `tam` değişkeninin saklandığı adresi tutacaktır. Bu durum aşağıdaki gibi tasvir edilir.



2.2. Neden Gösterici (Pointer) Kullanırız?

Gösterici, bellek alanındaki bir gözün adresinin saklandığı değişkendir. Göstericilere veriler (yani değişkenlerin içeriği) değil de, o verilerin bellekte saklı olduğu hücrenin başlangıç adresleri atanır. Kısaca gösterici adres tutan bir değişkendir. Göstericilerin C programlama dili içerisinde kullanılmasının sebepleri aşağıda özetlenmektedir.

- C programlama dilinin en güçlü özelliklerinden biridir.
- Göstericiler, işaretçiler ya da pointer adı da verilmektedir.
- Gösterici (pointer); içerisinde bellek adresi tutan değişkenlerdir.
- Şu ana kadar gösterilen tüm değişkenler tanımlandıkları türe göre bir değer saklarlar.
- Bu değere ulaşılmak istendiğinde doğrudan değişkenin ismi ile erişilebilir.
- Göstericiler ise değerin bulunduğu bellek hücresinin adresini tutmak sureti ile dolaylı olarak değere erişebilirler.

2.3. Call By Reference – Value Yaklaşımı

C program kodu yazılırken iki adet temel prensip üzerinden hareket edilmektedir. Bunlardan ilki değer odaklı çağırma (**call by value**) bir diğeri ise referans yani adres odaklı çalışma (**call by reference**) metotlarıdır. Değer odaklı çalışmada değişkenin değeri göz önüne alınarak kodlama yapılır ve bu değişkenin hafıza içerisinde nerede tutulduğu önemsenmezken referans odaklı çalışmada değişkenin hafıza içerisinde hangi adres ya da alanlarda tutulduğuna yani adresine göre işlem yapılmaktadır. Bir değişken tanımlandığında aşağıdakine benzer bir işlem yapılmaktadır.

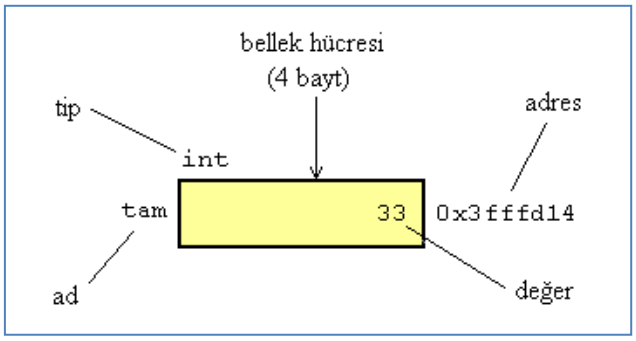
Örneğin `tam` adlı bir tamsayı değişkenini aşağıdaki gibi tanımladığımızı varsayalım:

```
int tam = 33;
```

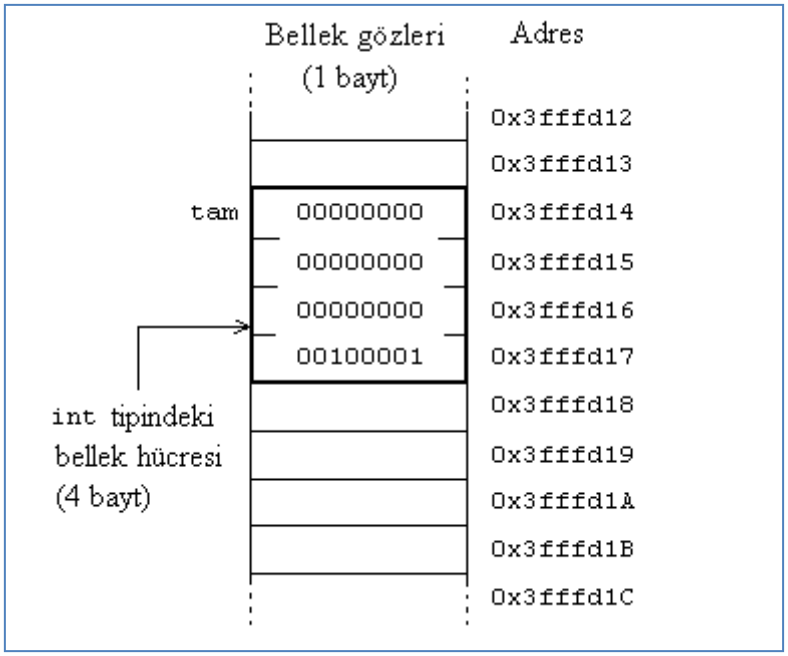
Bu değişken için, `int` tipinde bellekte (genellikle her biri 1 bayt olan 4 bayt büyüklüğünde) bir hücre ayrılır ve o hücreye 33 sayısı ikilik (binary) sayı sitemindeki karşılığı olan 4 baytlık (32 bitlik):

```
00000000 00000000 00000000 00100001
```

sayısı elektronik olarak yazılır. `tam` değişkenine ait dört temel özellik aşağıdaki gibi gösterilebilir:



Bellek adresleri genellikle onaltılık (hexadecimal) sayı sisteminde ifade edilir. `0x3fffd14` sayısı onluk (decimal) sayı sisteminde 67108116 sayına karşık gelir. Bunun anlamı, `tam` değişkeni, program çalıştığı sürece, bellekte 67108116. - 67108120 numaralı gözler arasındaki 4 baytlık hücreyi işgal edecek olmasıdır. Gerçekte, `int` tipindeki `tam` değişkeninin bellekteki yerleşimi ve içeriği (değeri) aşağıdaki gibi olacaktır.



2.4. & / * adres operatörleri

& işareti herhangi bir değişkenin adres değerine ulaşmak için kullanılmaktadır. * operatörü işaretçilerin tanımlanmasında ve bir işaretçinin referans aldığı adres içerisindeki değere ulaşılması için kullanılmaktadır. * ve & operatörleri birbirlerine ters bir işlem gerçekleştirmektedir. [3]

Aşağıda bu iki operatörün kullanımına örnek verelim.

4 0	12 1	5 2	15 3
6 4	9 5	13 6	11 7
14 8	10 9	2 10	1 11
0 12	3 13	7 14	8 15

Yukarıdaki tablo temsili olarak 4x4 bir hafıza yapısıdır. Bu yapı içerisinde sağ alt köşedeki rakamlar adresleri ifade ederken sol üst köşedeki değerler o adresleri kullanan değişkenlerin değerleridir. Aşağıda & ve * operatörlerini kullanımını örneklerle açıklayalım.

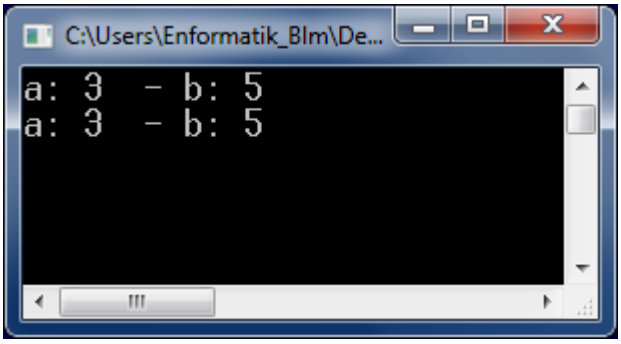
&9 = 5	içinde 9 değeri olan kutucuğun adresi 5'tir.
*10 = 2	adresi 10 olan kutucuğun içinde 2 değeri vardır.
(&(&(&(*3))))	3'te 15 var, içinde 15 olan 3'tür, içinde 3 olan 13'tür, içinde 13 olan 6'dır.
(&(&(*(*(&2))))	İçinde 2 olan kutucuk 10'dur, 10'un içinde 2 vardır, 2'nin içinde 5 var, 5'in olduğu kutucuk 2'dir, 2 10'un içindedir.

2.5. İşaretçilerin fonksiyonlar ile kullanılması

İşaretçi kavramının anlaşılması için verilebilecek en basit örneklerden biri takas fonksiyonu örneğidir. Ana fonksiyon içinde tamsayı tipinde a ve b ismindeki iki değişkenin değerleri birbiri ile değiştirilmek için takas isimli bir fonksiyon kullanılması istensin. Aşağıda bu takas işleminin işaretçi kullanılmadan ve kullanıldığında nasıl sonuçlar elde ettiğimizi görelim.

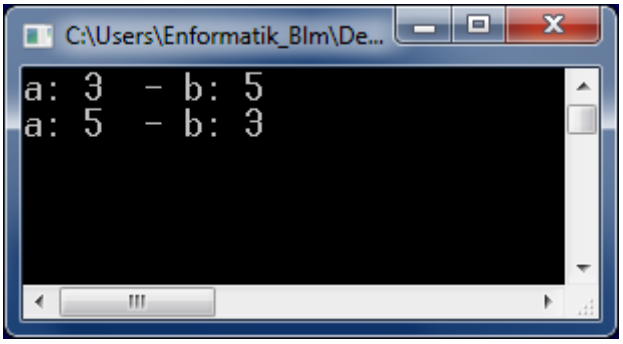
İşaretçi kullanılmadan yapılan takas fonksiyonu ve ana program içerisinde çağırılması;

```
#include<stdio.h>
#include<conio.h>
void takas(int x, int y);

main()
{
	int a = 3;
	int b = 5;
	printf("a: %d  - b: %d\n",a,b);
	takas(a,b);
	printf("a: %d  - b: %d\n",a,b);
	getch();
}

void takas(int x, int y)
{
	int yedek;
	yedek = x;
	x = y;
	y = yedek;
}
```



İşaretçi kullanılarak yapılan takas fonksiyonu ve ana program içinde çağırılması;

```
#include<stdio.h>
#include<conio.h>
void takas(int *x, int *y);

main()
{
	int a = 3;
	int b = 5;
	printf("a: %d  - b: %d\n",a,b);
	takas(&a,&b);
	printf("a: %d  - b: %d\n",a,b);
	getch();
}

void takas(int *x, int *y)
{
	int yedek;
	yedek = *x;
	*x = *y;
	*y = yedek;
}
```



2.1. Bölüm Çalışma Soruları

4 [0]	12 [1]	5 [2]	15 [3]
6 [4]	9 [5]	13 [6]	11 [7]
14 [8]	10 [9]	2 [10]	1 [11]
0 [12]	3 [13]	7 [14]	8 [15]

1-2-3. soruları yukarıdaki tabloya göre cevaplandırınız.

1. **(&(&(&(*(*(&3))))))** ifadesinin sayısal karşılığı nedir?
2. **(*(*(*(*10))))** ifadesinin sayısal karşılığı nedir?
3. **(&(&(&(&(&0)))))** ifadesinin sayısal karşılığı nedir?
4. Aşağıdaki programın ekran çıktısı ne olur?

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

main()
{
	int a = 3;
	int b = 5;
	int *aprt = &a;
	int *bptr = &*aprt;
	printf("a: %d  - b: %d\n",a,b);
	b = *bptr;
	printf("a: %d  - b: %d\n",a,b);
	getch();
}
```

3. SIRALAMA ALGORİTMALARI

3.1. Baloncuk Sıralama (Buble Sort)

Sıralama algoritmaları arasında yazılması en kolay olan, ama büyük dizilerde çok yavaş kalan bir sıralama yöntemidir. Zaman karmaşası (time complexity) O(n2) dir. Bu algoritma şöyle çalışır. Verilen dizinin öğelerini, veriliş sırasıyla yatay bir doğru boyunca dizelim.

1.Geçiş: Verilen dizinin (soldaki) ilk öğesi ikinci öğe ile karşılaştırılır. Bu karşılaştırmada, büyük olan öğe solda kalırsa, sağ yana geçirilmesi için, iki öğenin yerleri değiştirilir (takas, swap). Sonra büyük olan öğe, üçüncü öğe ile mukayese edilir. Büyük olan sağa geçer ve dördüncü öğe ile mukayese edilir. Bu işlem verilen dizinin son öğesine gelinene kadar devam eder. Bu geçişin sonunda, verilen dizinin en büyük terimi, en sağa yerleşmiş olur.

2.Geçiş: İlk geçişte en sağa yerleşen en büyük terim çıkarıldığında geriye kalan altdizine ilk geçişin aynısı uygulanır. Bu işlem alt dizinin en büyük öğesini seçer ve sağ uca yerleştirir. Bütün diziye bakacak olursak, verilen dizinin en büyük iki terimi en sağ uca kendi aralarında sıralanmış olarak yerleşir. Sıralama sağdan sola doğru azalan sıradadır ki bu soldan sağa doğru artan sırada olmasına denktir.

3.Geçiş: İlk iki geçişte en sağa yerleşen iki büyük terim çıkarıldığında geriye kalan altdizine ilk geçişin aynısı uygulanır. Bu işlem alt dizinin en büyük öğesini seçer ve sağ uca yerleştirir. Bütün diziye bakacak olursak, verilen dizinin en büyük üç terimi en sağ uca, sağdan sola doğru azalan sırada yerleşmiş olur.

n.Geçiş: Dizinin n terimi varsa, yukarıdaki gibi her geçişten sonra geriye kalan altdiziye ilk geçişte anlatılan yöntemi uygulayarak devam edelim. n geçişten sonra, dizinin terimleri sağdan sola doğru büyükten küçüğe doğru sıralanmış olacaktır. Bu ise, dizinin soldan sağa doğru artan sırada yerleşmiş olmasına denktir. [3]

Örnek:

74	57	3	7	95	-6	17

dizisini bubble sort algoritması ile sıralayalım. Aşağıdaki tablolarda, dizinin sağında kırmızı ile yazılan alt diziler, sağdan sola doğru azalan sırada sıralanmışlardır.

74	57	3	7	95	-6	17	
57	74	3	7	95	-6	17	
57	3	74	7	95	-6	17	
57	3	7	74	95	-6	17	
57	3	7	74	95	-6	17	
57	3	7	74	-6	95	17	
57	3	7	74	-6	17	**95**	
57	3	7	74	-6	17	95	*Birinci geçişin sonu*

Kaynak Kod :

```
#include<stdio.h>
#include<conio.h>
void dizi_gor(int a[8]);
main()
{
	int x[8]={8,7,6,5,4,3,2,1};
	int yedek;
	dizi_gor(x);
	getch();
	for(int i=0;i<7;i++)
	{
		for(int j=0;j<7;j++)
		{
			if(x[j]>x[j+1])
			{
				yedek = x[j+1];
				x[j+1]=x[j];
				x[j] = yedek;
			}
		}
		dizi_gor(x);
	}
	getch();
}
void dizi_gor(int a[8])
{
	printf("\n");
	for(int i=0;i<8;i++)
	{
		printf("\t%d",a[i]);
	}
}
```

3.2. Araya Eklemeli Sıralama (Insertion Sort)

Bu sıralama Bubble Sort algoritmasının iyileştirilmiş biçimidir. Zaman karmaşası (time complexity) $O(n^2)$ dir.

Bu algoritmayı açıklayan basit bir örnek verebiliriz.

Masada bir deste oyun kağıdı, sırasız ve sırtları yukarıya doğru duruyor olsun.

a. Desteden en üstteki kartı alalım. Onu masaya yüzü görünür biçimde koyalım. Tek kart olduğu için sıralı bir kümedir.

b. Sırasız destenin üstünden bir kart daha çekelim. Masadaki ilk çektiğimiz kart ile karşılaştıralım. Gerekirse yerlerini değiştirerek, çektiğimiz iki kartı küçükten büyüğe doğru sıralayalım.

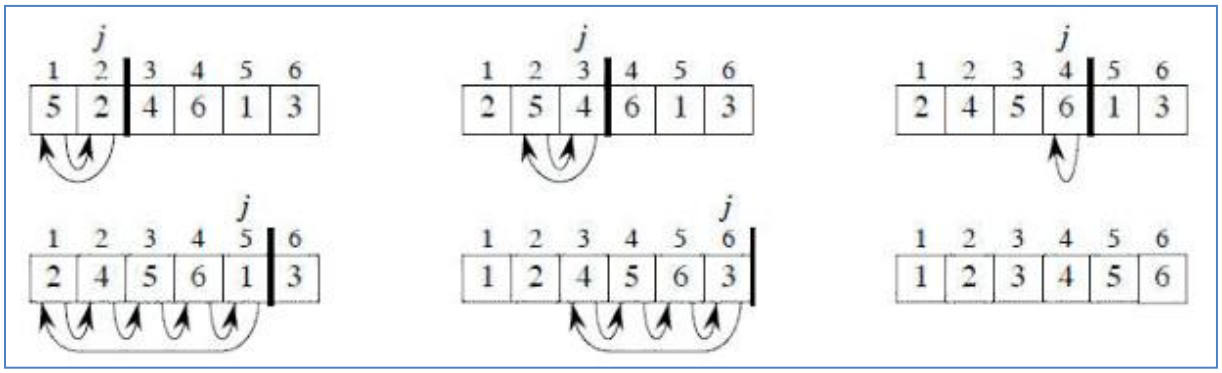
c. Sırasız destenin üstünden bir kart daha çekelim. Masaya sıralı dizilen iki kart ile karşılaştıralım. Gerekirse yerlerini değiştirerek çekilen üç kartı küçükten büyüğe doğru sıralayalım.

d. Bu işleme sırasız deste bitene kadar devam edelim. Sonunda, oyun kağıtlarını sıralamış oluruz. Herhangi bir adımda, kağıt destesi sıralı alt deste ve sırasız alt deste olmak üzere iki altkümeye ayrılmış durumdadır. Her adımda, sırasız desteden bir kart çekip, sıralı destedeki doğru yerine yerleştiriyoruz. Bu iş, sırasız deste bitene kadar devam ediyor.

Örnek:

5 2 4 6 1 3

dizisini insertion sort algoritması ile sıralayalım.



Kaynak Kod :

```
#include<stdio.h>
#include<conio.h>
void dizi_gor(int a[8]);
void insertsort(int x[], int n);
main()
{
	int a[] = {4,2,8,6,3,5,7,1};
	int dizi_boy = sizeof(a)/sizeof(int);
	dizi_gor(a);
	insertsort(a,dizi_boy);
	dizi_gor(a);
	getch();
}
void dizi_gor(int a[8])
{
	printf("\n");
	for(int i=0;i<8;i++)
	{
		printf("\t%d",a[i]);
	}
}
void insertsort(int x[], int n)
{
	int i,k,y;
	for(k=1; k<n; k++)
	{
		y=x[k];
		for(i=k-1; i>=0 && y<x[i]; i--)
		{
			x[i+1]=x[i];
		}
		x[i+1]=y;
	}
}
```

3.3. Seçerek Sıralama (Selection Sort)

Selection Sort algoritması $O(n^2)$ grubuna giren bir sıralama yöntemidir. Dolayısıyla büyük sayıda verileri sıralamak için elverişsizdir. Bunun yanında, bubble sort algoritmasındaki takas işlemlerinin çoğunu yapmadığı için, bubble sort algoritmasının iyileştirilmiş biçimi sayılır. Çünkü takas işlemlerinin sayısını $O(n^2)$ den $O(n)$ ye düşürür. Dolayısıyla daha etkilidir. Ancak, yaptığı mukayese işlemleri gene $O(n^2)$ düzeyindedir.

Algoritmanın çalışma ilkesi basittir.

1. Başlarken dizinin ilk öğesini en küçük öğe kabul eder. Tabii, bu kabul geçicidir. Sonra kalan terimler arasında daha küçükler varsa, onların en küçüğü olan terimle takas eder. O terimi en sola koyar; bu terim sıralama sonunda ilk terim olacaktır.

2. Diziden seçilen bu en küçük terimi çıkarınca, geri kalan alt dizine aynı yöntemi uygular. Alt diziden seçtiği en küçük öğeyi, ilk seçtiğinin sağına koyar. Dizinin sol ucunda iki terimli alt dizi küçükten büyüğe doğru sıralı bir alt dizi oluşturur.

3. Bu işleme kalan altdizinler bitene kadar devam edilir. Her adım başlarken, sol yanda sıralı bir alt dizi, sağ yanda sırasız bir alt dizi vardır. Her adımda sıralı dizi bir terim artar, sırasız dizi bir terim azalır. Sağ yandaki sırasız alt dizi bitince, sıralama işlemi biter.

Örnek:

int [] a = {3,17,86,-9,7,-11,38} dizisi verilsin. Görselliği sağlamak için, bu diziyi

3 17 86 -9 7 -11 38

dizisi biçiminde yazalım. Bu dizinin öğelerini selection sort algoritması ile sıralayacağız. Dizinin (array) indislerinin 0 dan başladığını anımsayınız.

a[0] = 3,…, a[6] = 38 dir.

Başlangıçta bütün dizi sırasızdır. Dizinin ilk öğesini seçip, tek öğeli (sıralı) bir alt dizi oluşturabiliriz. Geçici olarak, verilen dizinin ilk öğesini enküçük öğe (minimal) imiş gibi kabul edelim. Sonra mukayese ile daha küçük terim olup olmadığını araştıracağız. Ondan daha küçükler varsa, onların en küçüğünü a[0] ile takas edeceğiz. Böylece verilen dizinin en küçük terimi en sola yerleşir.

1.Aşama

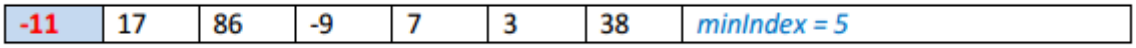
Başlangıçta a[0] = 3 olduğu için minimal = 3 olur. Bu eşitliği sağlayan indise minIndex diyelim. İlk değeri minIndex = 0 dır.

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
3	17	86	-9	7	-11	38

Sonra minimal 3 sayısını, sırayla dizinin öteki terimleriyle karşılaştırarak, 3 den daha küçük öğe olup olmadığını, varsa onların en küçüğünün hangisi olduğunu arayalım. 17 ve 86 terimleri koşulu sağlamaz; onları atlıyoruz. Üçüncü a[3] terimi ile mukayese yapınca $-9 < 3$ olduğunu görüyoruz. Bu durumda daha küçük olan -9 teriminin indisini minIndex yapıyoruz.

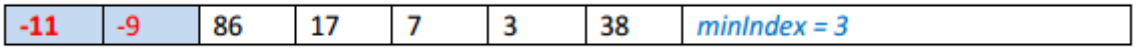
minIndex = 3

Bu andan sonra minimal öğemiz 3 değil -9 olmuştur. Ondan sonraki terimleri -9 ile karşılaştıracağız. 7 koşulu sağlamaz, onu atlıyoruz. Beşinci a[5] teriminde -11 < -9 olduğu için, minIndex = 5 olur. Bu aşamada minimal öğe indisi 5 olan -11 öğesidir. Kalan 38 terimini -11 ile mukayese ediyor ve koşulu sağlamadığını görüyoruz. O halde dizinin en küçük öğesi indisi minIndex = 5 olan -11 öğesidir. Dolayısıyla, 3 ile -11 öğelerinin yerlerini değiştiriyoruz (takas işlemi).



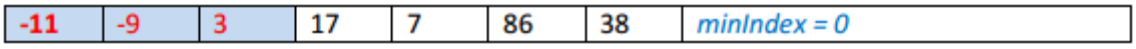
2.Aşama

Bu aşamada dizi sıralı {-11} ve sırasız {17, 86, -9, 7, 3, 38} olmak üzere iki altdiziye ayrılmıştır. Şimdi sırasız altdiziye yukarıdaki seçme algoritmasını uygulayarak, enküçük öğesini seçebiliriz. Bunun -9 olacağını görüyoruz. Alt dizinin ilk öğesi olan 17 terimi ile en küçük öğesi olan -9 terimlerinin yerlerini değiştiriyoruz (takas). Sonunda, -9 terimini sıralı alt diziye ekliyoruz:



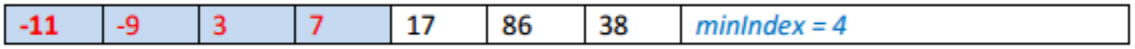
3.Aşama

Bu aşamada dizi sıralı {-11, -9} ve sırasız {86, 17, 7, 3, 38} olmak üzere iki altdiziye ayrılmıştır. Şimdi sırasız altdiziye seçme algoritmasını uygulayarak, enküçük öğesini seçebiliriz. Bunun 3 olacağını görüyoruz. Alt dizinin ilk öğesi olan 86 terimi ile en küçük öğesi olan 3 terimlerinin yerlerini değiştiriyoruz (takas). Sonunda, 3 terimini sıralı alt diziye ekliyoruz:



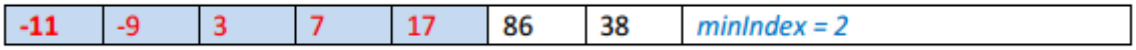
4.Aşama

Bu aşamada dizi sıralı {-11, -9, 3} ve sırasız {17, 7, 86, 38} olmak üzere iki altdiziye ayrılmıştır. Şimdi sırasız alt diziye seçme algoritmasını uygulayarak, enküçük öğesini seçebiliriz. Bunun 7 olacağını görüyoruz. Alt dizinin ilk öğesi olan 17 terimi ile en küçük öğesi olan 7 terimlerinin yerlerini değiştiriyoruz (takas). Sonunda, 7 terimini sıralı alt diziye ekliyoruz:



5.Aşama

Bu aşamada dizi sıralı {-11, -9, 3, 7} ve sırasız {17, 86, 38} olmak üzere iki alt diziye ayrılmıştır. Şimdi sırasız alt diziye seçme algoritmasını uygulayarak, en küçük öğesini seçebiliriz. Bunun 17 olacağını görüyoruz. Alt dizinin ilk öğesi zaten 17 terimidir. Dolayısıyla bir takas işlemi gerekmiyor. Sonunda, 17 terimini sıralı alt diziye ekliyoruz:

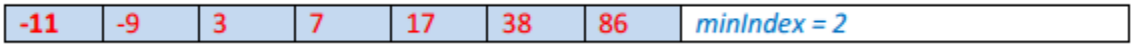


6.Aşama

Bu aşamada dizi sıralı {-11, -9, 3, 7, 17} ve sırasız {86, 38} olmak üzere iki alt diziye ayrılmıştır. Şimdi sırasız alt diziye seçme algoritmasını uygulayarak, en küçük öğesini seçebiliriz. Bunun 38 olacağını görüyoruz. Alt dizinin ilk öğesi olan 86 terimi ile en küçük öğesi olan 38 terimlerinin yerlerini değiştiriyoruz (takas). Sonunda, 38 terimini sıralı alt diziye ekliyoruz:

7.Aşama

Bu aşamada dizi sıralı {-11, -9, 3, 7, 17, 38} ve sırasız {86} olmak üzere iki altdiziye ayrılmıştır. Sırasız dizi tek öğeli olduğu için, en küçük öğesi kendisidir. 86 terimini sıralı alt diziye ekliyoruz:



Kaynak Kod :

```
void selectionSort(int dizi[],int n)
{
 int yedek;
 int minIndex;
 for(int i=0; i<n-1; i++)
 {
 minIndex=i;
 for(int j=i; j<n; j++)
 {
 if (dizi[j] < dizi[minIndex]) minIndex=j;
 }
 temp=dizi[i];
 dizi[i]=dizi[minIndex];
 dizi[minIndex]=yedek;
 }
}
```

4. KAYNAKÇA

[1] D. A. Bingül, "http://www1.gantep.edu.tr/," Nisan 2011. [Online].

[2] F. DEĞİRMENCİOĞLU, "http://fdegirmencioglu.com/," 2013. [Online].

[3] Anonim, "http://www.cplusplus.com/doc/tutorial/pointers/," 2013. [Online].

[4] T. Karaçay, "http://www.baskent.edu.tr," 2012. [Online]. Available: http://www.baskent.edu.tr/~tkaracay/etudio/ders/prg/dataStructures/sorting.