BLM212 Veri Yapıları

Recursion
(Özyineleme)

2021-2022 Güz Dönemi

Tekrarlı algoritmalar yazmak için iki yaklaşım:

- **Iteration** (yineleme, tekrarlama, iterasyon)
- **Recursion** (Özyineleme)

➢**Özyineleme** (**Recursion**), bir algoritmanın kendisini çağırdığı, tekrarlı bir işlemdir.

- ➢ **Bir problemi, kendisinin daha küçük versiyonlarına indirgeyerek çözme süreci**

➢Özyineleme, bir alt program veya fonksiyonun kendisini çağıracağı şekilde düzenlenir.

Factorial – a case study

- Pozitif bir sayının faktöriyeli, 1’den o sayıya kadar olan sayıların çarpımıdır:

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n = \prod_{i=1}^{n} i$$

Factorial: Iterative Algorithm

$$\text{Factorial}(n) = \begin{bmatrix} 1 & \text{if } n = 0 \\ n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1 & \text{if } n > 0 \end{bmatrix}$$

FIGURE 2-1 Iterative Factorial Algorithm Definition

Tekrarlı bir algoritma iteratif olarak tanımlanmış ise, kendisini değil sadece algoritma parametrelerini ister (ihtiyaç duyar).

Factorial (4) = 4 x 3 x 2 x 1 = 24

Factorial: Recursive Algorithm

$$\text{Factorial}(n) = \begin{bmatrix} 1 & \text{if } n = 0 \\ n \times (\text{Factorial}(n-1)) & \text{if } n > 0 \end{bmatrix}$$

FIGURE 2-2 Recursive Factorial Algorithm Definition

Tekrarlı bir algoritma özyinelemeyi kullanıyor ise, algoritma adı tanımının içinde mutlaka geçiyordur.

Factorial (4) = 4 x Factorial(3)
= 4 x 3 x Factorial(2)
=4 x 3 x 2 x Factorial(1) = 24

Recursion: Temel fikir

- Bir problemin özyinelemeli çözümü **iki yönlü** yolculuk içerir:
  - İlk önce problemi yukarıdan aşağıya doğru ayrıştırırız
  - Sonra problemi aşağıdan yukarı doğru çözeriz.

Factorial (3): Ayrıştırma (Decomposition) ve çözüm

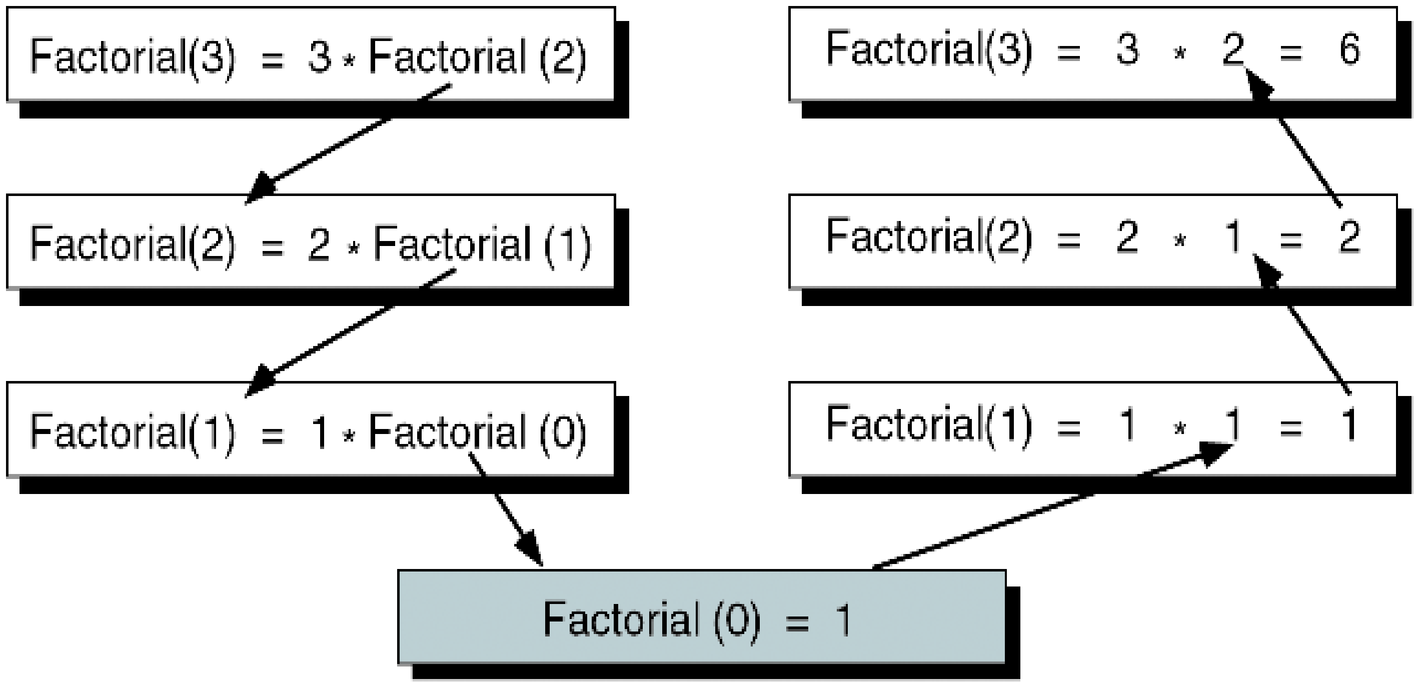


FIGURE 2-3 Factorial (3) Recursively

ALGORITHM 2-1 Iterative Factorial Algorithm

```
Algorithm iterativeFactorial (n)
Calculates the factorial of a number using a loop.
   Pre  n  is the number to be raised factorially
   Post n! is returned
1  set i to 1
2  set factN to 1
3  loop (i <= n)
   1  set factN to factN * i
   2  increment i
4  end loop
5  return factN
end iterativeFactorial
```

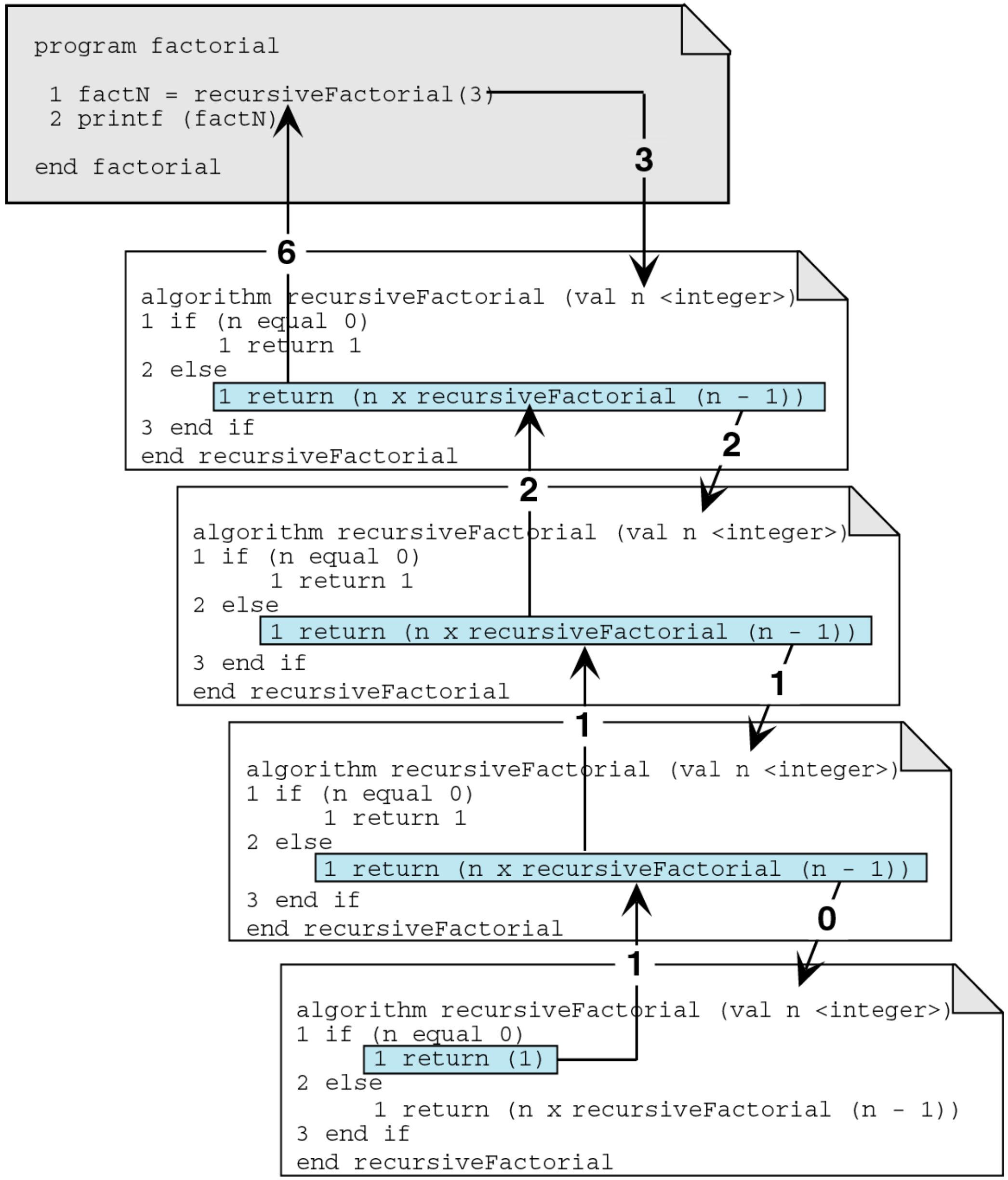
ALGORITHM 2-2 Recursive Factorial

```
Algorithm recursiveFactorial (n)
Calculates factorial of a number using recursion.
   Pre    n  is the number being raised factorially
   Post   n! is returned
1 if (n equals 0)
   1  return 1
2 else
   1  return (n * recursiveFactorial (n - 1))
3 end if
end recursiveFactorial
```

Özyineleme nasıl çalışır?

- Bir program bir alt programı çağırdığında, güncel modül işlemeyi askıya alır ve çağrılan alt program, programın kontrolünü ele geçirir.

- Alt program işlemlerini tamamladığında ve onu çağıran modüle geri döndüğünde
  - parametrelerin değeri çağrıdan önce ve sonra aynı olmalıdır.



Calling a recursive algorithm

Özyinelemeli algotirma dizaynı

- Özyinelemeli bir algoritmanın her çağrısı ya sorunun bir bölümünü çözer ya da sorunun boyutunu azaltır.

- Çözümün genel kısmı rekursif çağrıdır. Her rekursif çağrıda problemin boyutu azaltılır.

Özyinelemeli algotirma dizaynı

- Problemi “çözen” ifadeye **temel durum** (base case) denir.
- Her rekursif algoritmada mutlaka bir temel durum vardır.
- Algoritmanın geri kalanı **genel durum** (general case) olarak adlandırılır. Genel durum problemin boyutunu azaltmak için gereken mantığı içerir.

Özyinelemeli algotirma dizaynı

- Temel duruma (base case) ulaşıldığında çözüm başlar.
- Şimdi cevabın bir kısmını biliyoruzdur ve bu kısmı bir sonraki daha genel ifadeye geri döndürebiliriz.
- Bu, bir sonraki genel durumu (general case) çözmemizi sağlar.
- Her genel durumu sırayla çözdüğümüz için, nihayetinde en genel durumu yani orijinal problemi çözene kadar, sonraki daha üst genel durumu çözebiliriz.

Özyinelemeli algotirma dizaynı

- Özyinelemeli bir algoritma dizayn kuralları

  1. İlk olarak temel durum (*base case*) belirlenir
  2. Daha sonra genel durum (*general case*) belirlenir.
  3. Temel ve genel durum bir algoritmada birleştirilir.

Özyinelemeli algotirma dizaynı

- Her rekursif çağrı, sorunun boyutunu azaltmalı ve **temel duruma** (base case) getirmelidir.

- Temel duruma ulaşıldığında, özyinelemeli algoritmaya çağrı yapılmadan sonlandırılmalıdır;
  - yani, bir geri dönüş gerçekleştirmelidir.

Özyineleme Sınırlamaları

- Özyinelemeli çözümler, çağrıları kullandıkları için (hem **zaman** hem de **bellek**) yoğun bir ek yük içerebilir.

- Her bir çağrının icrası zaman alır.

- Bu nedenle özyinelemeli bir algoritma genellikle özyinelemesiz uygulamasından daha yavaş çalışır.

Özyineleme Sınırlamaları

- Aşağıdaki sorulardan herhangi birinin cevabı hayır ise, özyineleme kullanılmamalıdır:
  - Algoritma veya veri yapısı özyineleme için uygun mu?
  - Özyinelemeli çözüm daha kısa ve daha anlaşılır mı?
  - Özyinelemeli çözüm kabul edilebilir zaman ve bellek alanı sınırları dahilinde çalışıyor mu?
- Genel bir kural olarak, özyinelemeli algoritmalar sadece verimlilikleri/karmaşıklıkları (efficiency) logaritmik olduğunda etkili bir şekilde kullanılmalıdır.

Not: **Özyineleme**, algoritma özyinelemeyi destekleyen bir veri yapısını kullandığında, en iyi çalışır.

Recursive Definitions

- **Recursion** (Özyineleme)
  - Bir problemi, kendisinin daha küçük versiyonlarına indirgeyerek çözme süreci
- Örnek: factorial problem
  - 5!
    - 5 x 4 x 3 x 2 x 1 =120
  - Eğer n negatif değilse
    - Factorial of *n* (*n*!) şu şekilde tanımlanır

$$0! = 1 \qquad \text{(Equation 6-1)}$$

$$n! = n \times (n-1)! \quad \text{if} \quad n > 0 \qquad \text{(Equation 6-2)}$$

Recursive Definitions (cont'd.)

- **Direct solution** (Equation 6-1)
  - Denklemin sağ tarafı faktör notasyonu içermez
- **Recursive definition**
  - Bir şeyin kendisinin daha küçük bir versiyonuyla tanımlandığı bir tabir
- **Base case** (Equation 6-1)
  - Çözümün doğrudan elde edildiği durumdur
- **General case** (Equation 6-2)
  - Çözümün dolaylı olarak özyineleme kullanılarak elde edildiği durumdur

Recursive Definitions (cont'd.)

- Özyineleme anlayışı
  - Her rekursif tanım bir (veya daha fazla) temel durum içermelidir
  - Genel durum en sonunda bir temel duruma düşürülmeli
  - Temel durum özyinelemeyi durdurur
- **Recursive algorithm**
  - Problemi kendi küçük versiyonlarına indirgeyerek probleme çözüm bulur
- **Recursive function**
  - Kendisini çağıran fonksiyon

```
int fact(int num)
{
    if (num == 0)
        return 1;
    else
        return num * fact(num - 1);
}
```

Recursive Definitions (cont'd.)

- Rekursif fonksiyon-önemli bilgiler
  - Özyinelemeli fonksiyon sınırsız sayıda kopyaya sahip olabilir (**mantıksal olarak**)
  - Rekursif fonksiyona yapılan her çağrı kendi…
    - **Code**, **set of parameters**, **local variables**
  - Belirli bir rekursif çağrı tamamlandıktan sonra
    - Kontrol çağıran ortama (önceki çağrıya) geri döner
    - Kontrol bir önceki çağrıya geri dönmeden önce güncel (rekursif) çağrı tamamen gerçekleştirilmelidir.
    - Bir önceki çağrıda icra, rekursif çağrıyı hemen takip eden noktadan itibaren başlar

Recursive Definitions (cont'd.)

- **Infinite recursion** (Sonsuz özyineleme)
  - Her rekursif çağrı başka bir rekursif çağrı ile sonuçlanırsa oluşur.
  - (Teoride) sonsuza kadar icra eder
  - Rekursif foksiyonlar için çağrı gereksinimleri
    - Yerel değişkenler ve parametreler için sistem belleği
    - Çağırana geri dönüş bilgilerini kaydetme
  - Sistem belleğinin sınırlı oluşu…
    - Sistem belleği tükenene kadar icra edilir
    - Sonsuz rekursif fonksiyonun anormal sonlanmasına neden olur

Lab Uygulaması 1: C'de iç içe en fazla kaç kez rekursif çağrı yapılabilir? Üst sınır nedir? Bunu test eden programı yazınız.

Recursive Definitions (cont'd.)

- Rekursif fonksiyon dizayn gereksinimleri
  - Problemin gereksinimlerini anlamak
  - Sınır koşullarını belirleme
  - Her bir temel duruma doğrudan çözüm sağlayarak temel durumları tanımlama
  - Her genel duruma kendisinin küçük versiyonları bakımından çözüm sağlayarak genel durumları tanımlama

Örnek Problem

- Klavyeden veri okuduğumuzu ve verileri ters sırada yazdırmamız gerektiğini varsayalım.

- Listeyi tersten yazdırmanın en kolay yolu, özyinelemeli bir algoritma yazmaktır.

Çözüm

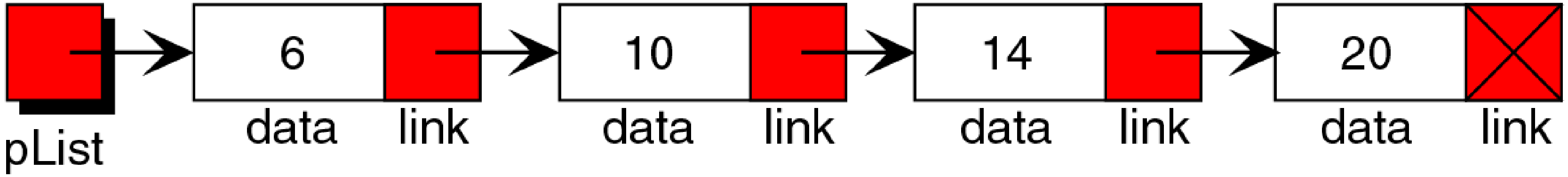
- **Listeyi tersten yazdırmak** için öncelikle tüm verileri okumamız gerektiği aşikardır.
  - Son veriyi okuduktan sonra ekrana basarsak - yani, özyinelemeden çıktığımızda ekrana basarsak - ters sırayla basarız.
- Bundan dolayı, temel durum (base case): son elemanı (veri parçasını) okumuş olmak
- Benzer bir şekilde genel durum (general case) ise sıradaki elemanı okumak

Implementation

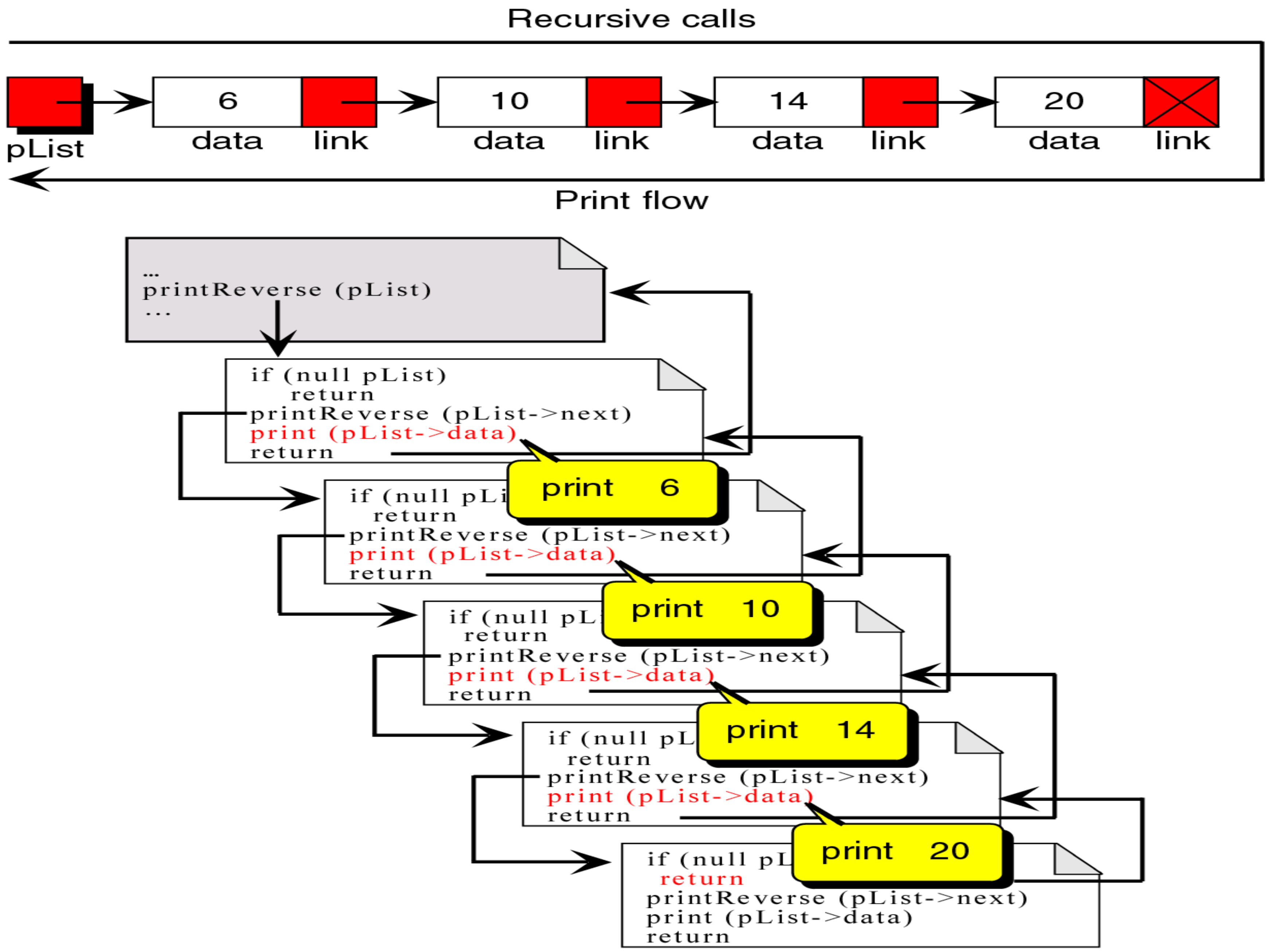
ALGORITHM 2-3 Print Reverse

```
Algorithm printReverse (data)
Print keyboard data in reverse.
   Pre  nothing
   Post data printed in reverse
1  if (end of input)
   1  return
2  end if
3  read data
4  printReverse (data)
Have reached end of input: print nodes
5  print data
6  return
end printReverse
```

Reverse a Linked List



Lab Uygulaması 2: Daha önce oluşturduğumuz düğüm veri yapısını (Node) kullanarak elemanlarını klavyeden girdiğiniz bir bağlı listeyi özyinelemeli fonksiyon kullanarak elemanlarnı tersi sırada ekrana yazdırınız.



Recursive calls
6
10
14
20
pList
data
link
Print flow
...
printReverse (pList)
...
if (null pList)
return
printReverse (pList->next)
print (pList->data)
return
print 6
print 10
print 14
print 20

Analysis

- Algoritma veya veri yapısı özyineleme için uygun mu?
  - Klavyeden okunan veriler gibi bir liste doğal olarak **özyinelemeli yapı değildir**. Dahası, algoritma logaritmik bir algoritma değildir.
- Özyinelemeli çözüm daha kısa ve daha anlaşılır mı?
  - Evet
- Özyinelemeli çözüm kabul edilebilir zaman ve bellek alanı sınırları dahilinde çalışıyor mu?
  - Bir listede gezinmedeki yineleme sayısı algoritma lineer bir verimliliğe sahip olduğundan oldukça büyük olabilir - O (n).

Örnek Problem

- **İki sayının en büyük ortak bölenini** (*greatest common divisor*-GCD) belirleyin.
- Öklid algoritması: GCD (a, b), aşağıdaki formülden özyinelemeli bir şekilde bulunabilir.

$$GCD(a,b) = \begin{cases} a & \text{if } b=0 \\ b & \text{if } a=0 \\ GCD(b, a \bmod b) & \text{otherwise} \end{cases}$$

Pseudocode Implementation

ALGORITHM 2-4 Euclidean Algorithm for Greatest Common Divisor

```
Algorithm gcd (a, b)
Calculates greatest common divisor using the Euclidean algo-
rithm.
   Pre  a and b are positive integers greater than 0
   Post greatest common divisor returned
1 if (b equals 0)
   1  return a
2 end if
3 if (a equals 0)
   2  return b
4 end if
5 return gcd (b, a mod b)
end gcd
```

C implementation

PROGRAM 2-1 GCD Driver

```c
/* This program determines the greatest common divisor
   of two numbers.
      Written by:
      Date:
*/
#include <stdio.h>
#include <ctype.h>

// Prototype Statements
int gcd (int a, int b);

int main (void)
{
// Local Declarations
   int  gcdResult;

// Statements
   printf("Test GCD Algorithm\n");

   gcdResult = gcd (10, 25);
   printf("GCD of 10 & 25 is %d", gcdResult);
   printf("\nEnd of Test\n");
   return 0;
}  // main
```

PROGRAM 2-1 GCD Driver (continued)

```
/* ================= gcd =================
   Calculates greatest common divisor using the
   Euclidean algorithm.
      Pre  a and b are positive integers greater than 0
      Post greatest common divisor returned
*/
int gcd (int a, int b)
{
   // Statements
   if (b == 0)
      return a;
   if (a == 0)
      return b;
   return gcd (b, a % b);
}  // gcd
```

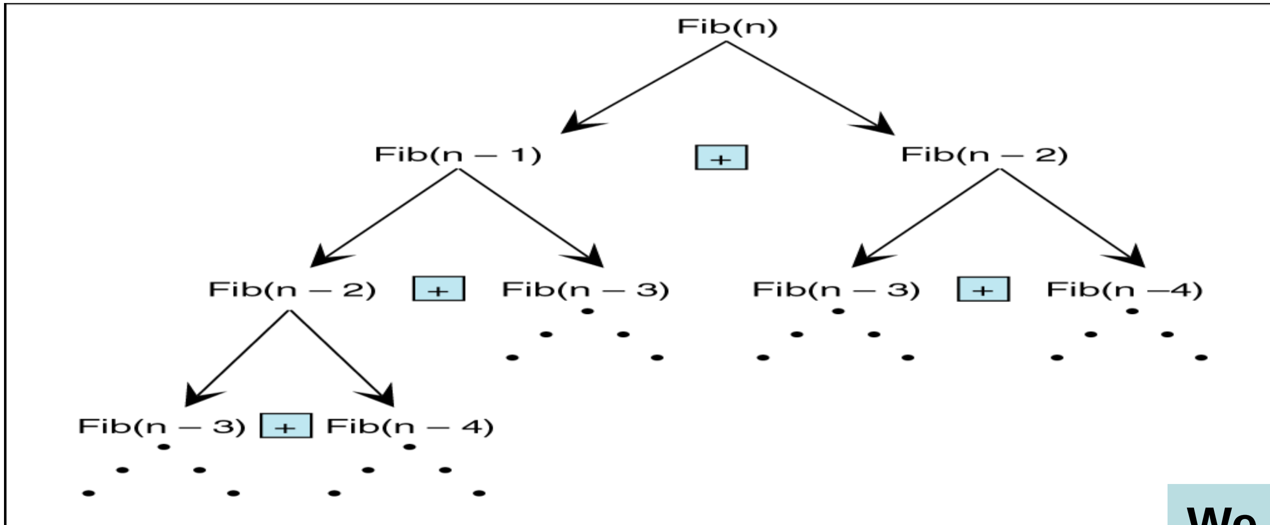
```
Results:
Test GCD Algorithm
GCD of 10 & 25 is 5
End of Test
```

Örnek Problem

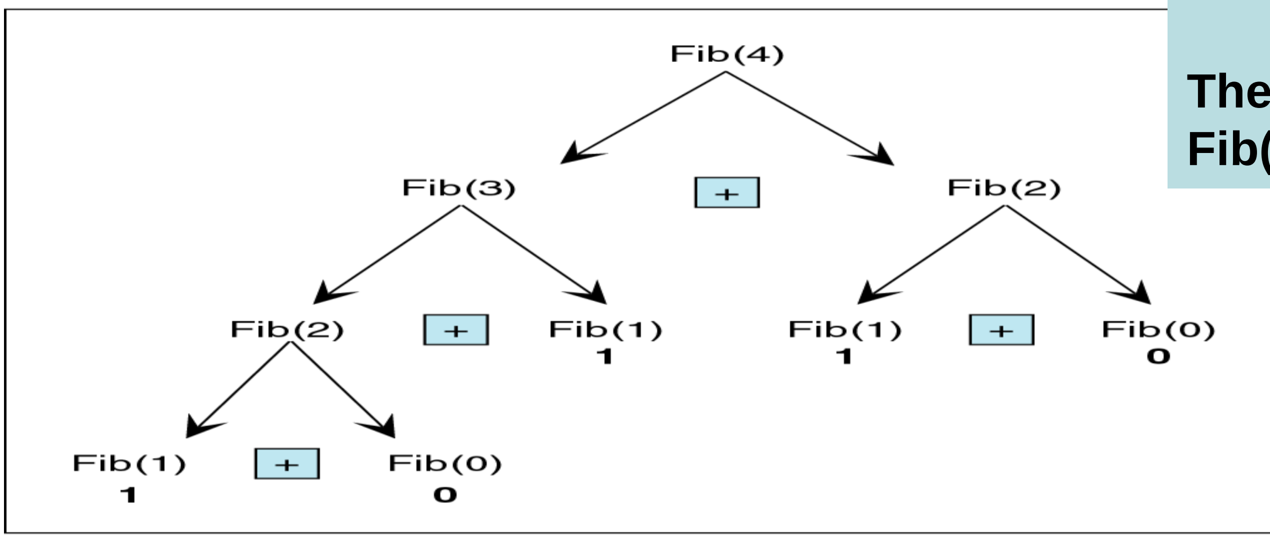
- **Fibonacci sayıları serisinin üretilmesi**
  - Takip eden her sayı önceki iki sayının toplamına eşittir.
  - Klasik Fibonacci serisi: 0, 1, 1, 2, 3, 5, 8, 13, …
  - n sayıdan oluşan seri aşağıdaki rekursif formül kullanılarak hesaplanabilir.

$$Fibonacci(n) = \begin{cases} 0 & \text{if } n=0 \\ 1 & \text{if } n=1 \\ Fibonacci(n-1) + Fibonacci(n-2) & \text{otherwise} \end{cases}$$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...



(a) Fib(n)



(b) Fib(4)

We can generalize it:
Given: Fib(0)=0
Fib(1)=1
Then:
Fib(n)=Fib(n-1)+Fib(n-2)...

PROGRAM 2-2 Recursive Fibonacci Series

```
/* This program prints out a Fibonacci series.
      Written by:
      Date:
*/
#include <stdio.h>

// Prototype Statements
   long fib (long num);

int main (void)
{
// Local Declarations
   int seriesSize = 10;

// Statements
   printf("Print a Fibonacci series.\n");

   for (int looper = 0; looper < seriesSize; looper++)
       {
        if (looper % 5)
           printf(", %8ld", fib(looper));
        else
           printf("\n%8ld", fib(looper));
       } // for
   printf("\n");
   return 0;
}  // main
```

PROGRAM 2-2 Recursive Fibonacci Series (Continued)

```

/* ================= fib =================
   Calculates the nth Fibonacci number
      Pre  num identifies Fibonacci number
      Post returns nth Fibonacci number
*/
long fib (long num)
{
// Statements
   if (num == 0 || num == 1)
      // Base Case
      return num;
   return (fib (num - 1) + fib (num - 2));
}  // fib
```

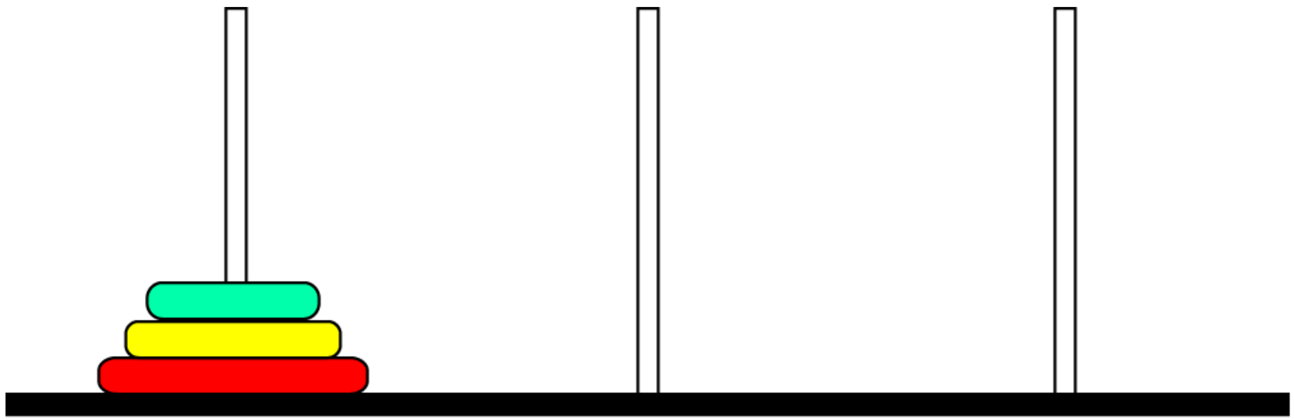
```
Results:
Print a Fibonacci series.

       0,         1,         1,         2,         3
       5,         8,        13,        21,        34
```

fib(*n*)	Calls	fib(n)	Calls
1	1	11	287
2	3	12	465
3	5	13	753
4	9	14	1219
5	15	15	1973
6	25	20	21,891
7	41	25	242,785
8	67	30	2,692,573
9	109	35	29,860,703
10	177	40	331,160,281

TABLE 2-1 Fibonacci Calls

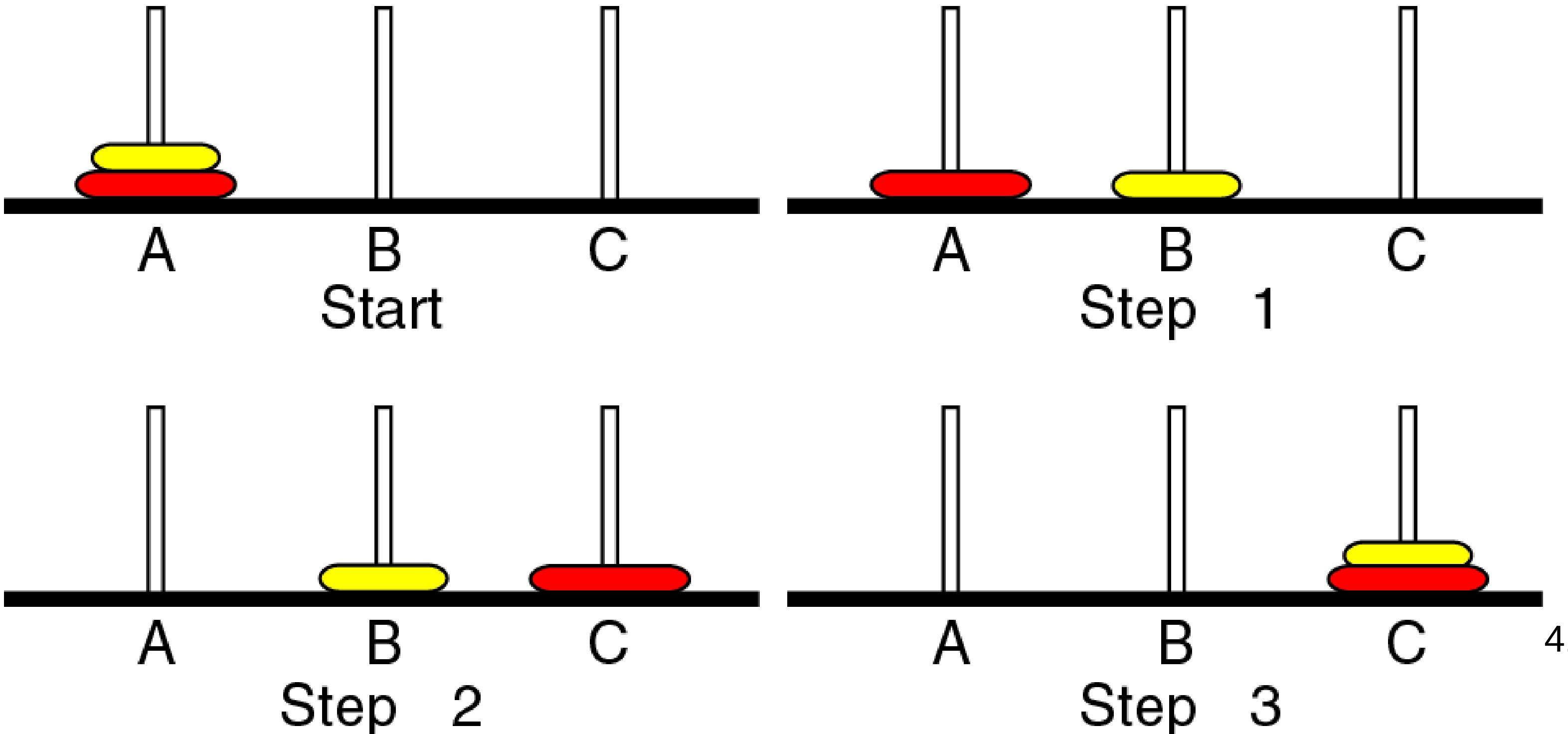
Örnek Problem

- **Hanoi Kuleleri (**The Towers of Hanoi**)**
  1. Aynı anda sadece bir tane disk hareket ettirilebilir. Daha büyük bir disk asla daha küçük bir disk üzerine istiflenmemelidir.
  2. Disklerin ara (*auxiliary*) depolama işlemi için sadece bir yardımcı iğne kullanılabilir.



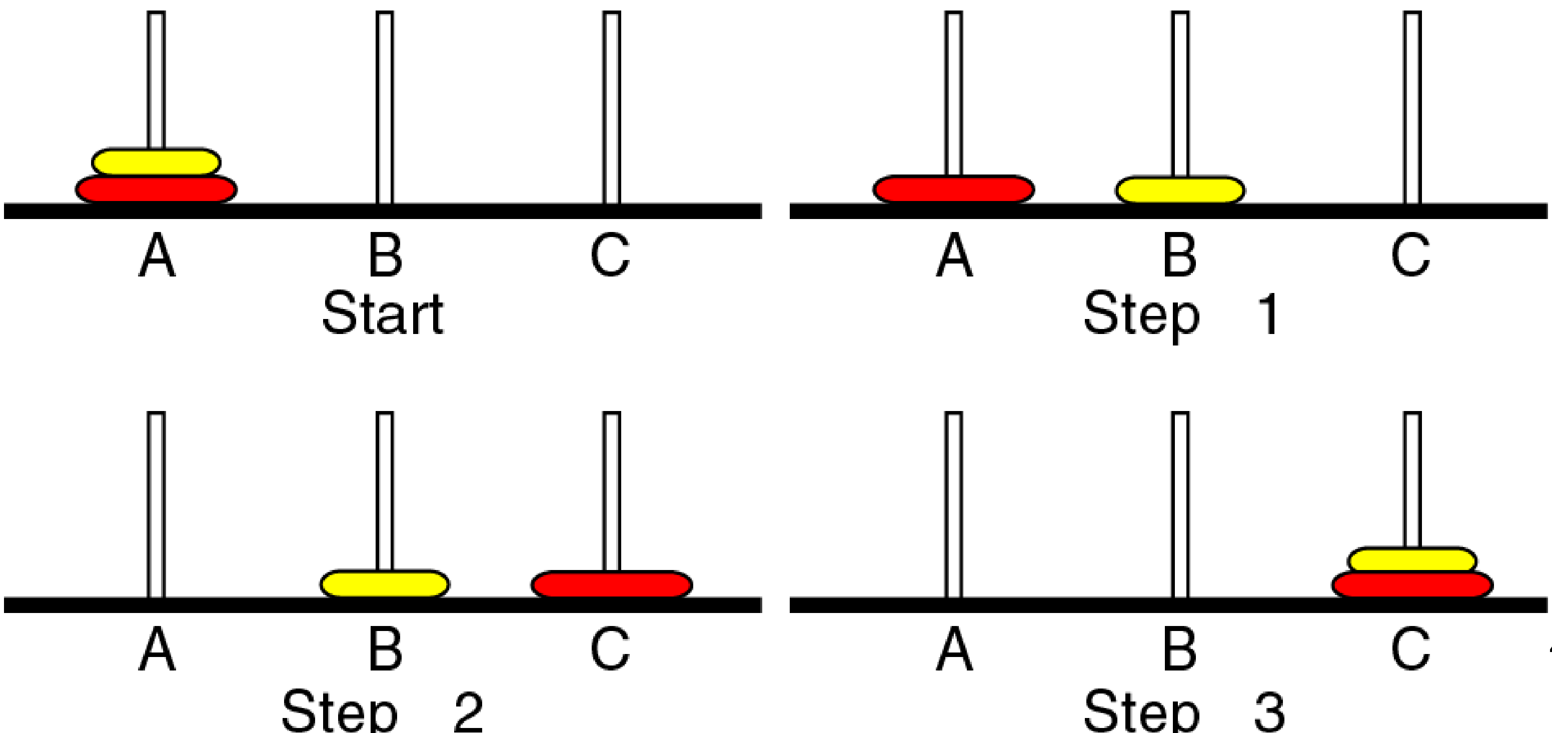
The Towers of Hanoi

Case 2 (İki disk)



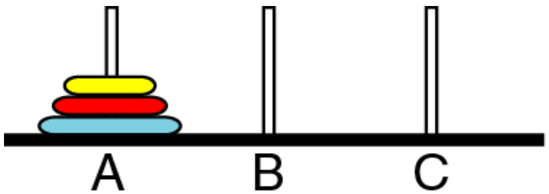
The Towers of Hanoi
Case 2 (İki disk)

1. Move one disk to auxiliary needle.
2. Move one disk to destination needle.
3. Move one disk from auxiliary to destination needle.



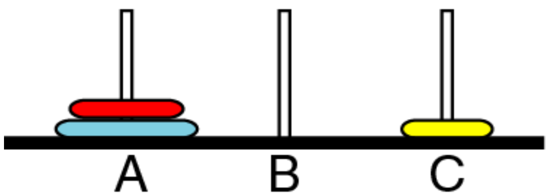
The Towers of Hanoi

Case 3 (Üç disk)

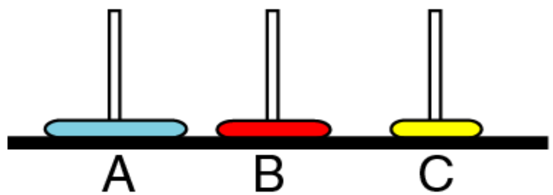


Start

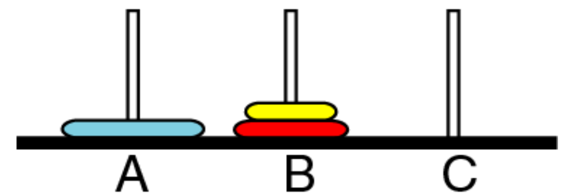
1. Move two disks from source to auxiliary needle. (Step 1-3)
2. Move one disk from source to destination needle. (Step 4)
3. Move two disks from auxiliary to destination needle. (Step 5-7)



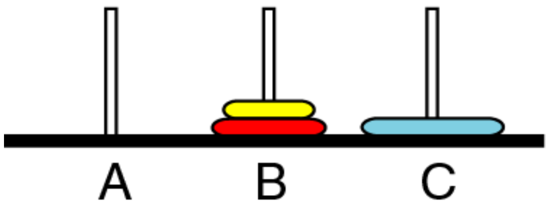
Step 1



Step 2

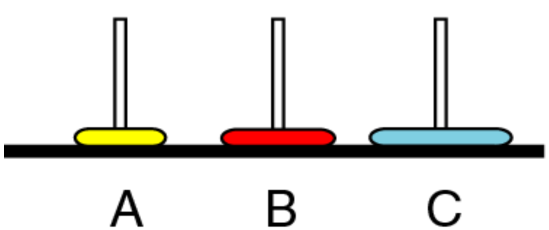


Step 3

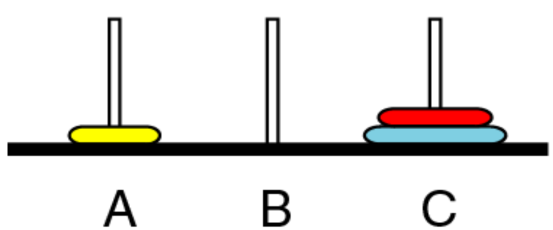


Step 4

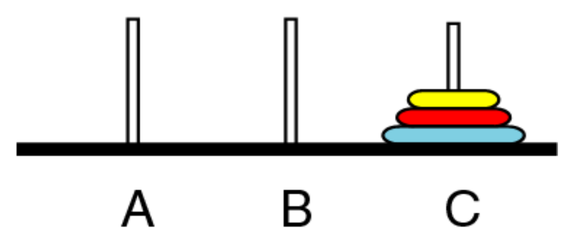
Move one disk from source to destination.



Step 5



Step 6



Step 7

The Towers of Hanoi

Problemi genelleştirecek olursak;

1.Move n-1 disks from source to auxiliary. **General Case**

2.Move one disk from source to destination. **Base Case**

3.Move n-1 disks form auxiliary to destination. **General Case**

```
1. Call Towers (n - 1, source, auxiliary, destination)
2. Move one disk from source to destination
3. Call Towers (n - 1, auxiliary, destination, source)
```

Bu sefer kaynak (source) iğne, yardımcı (auxiliary) iğne olur.

The Towers of Hanoi

algorithm towers (val **disks** <integer>, val **source** <character>, val **dest** <character>, val **auxiliary** <character>, **step** <integer>)

Recursively move one disk from source to destination.

Pre: The tower consists of integer disks

Source, destination and auxilary towers given.

Post: Steps for moves printed

1 print(“Towers :”, disks, source, dest, auxiliary)

2 if (disks =1)

  1 print(“Step”, step, “Move from”, source, “to”, dest)

  2 step = step + 1

3 else

  1 towers(disks – 1, source, auxiliary, dest, step)

  2 print(“Step”, step, “Move from”, source, “to”, dest)

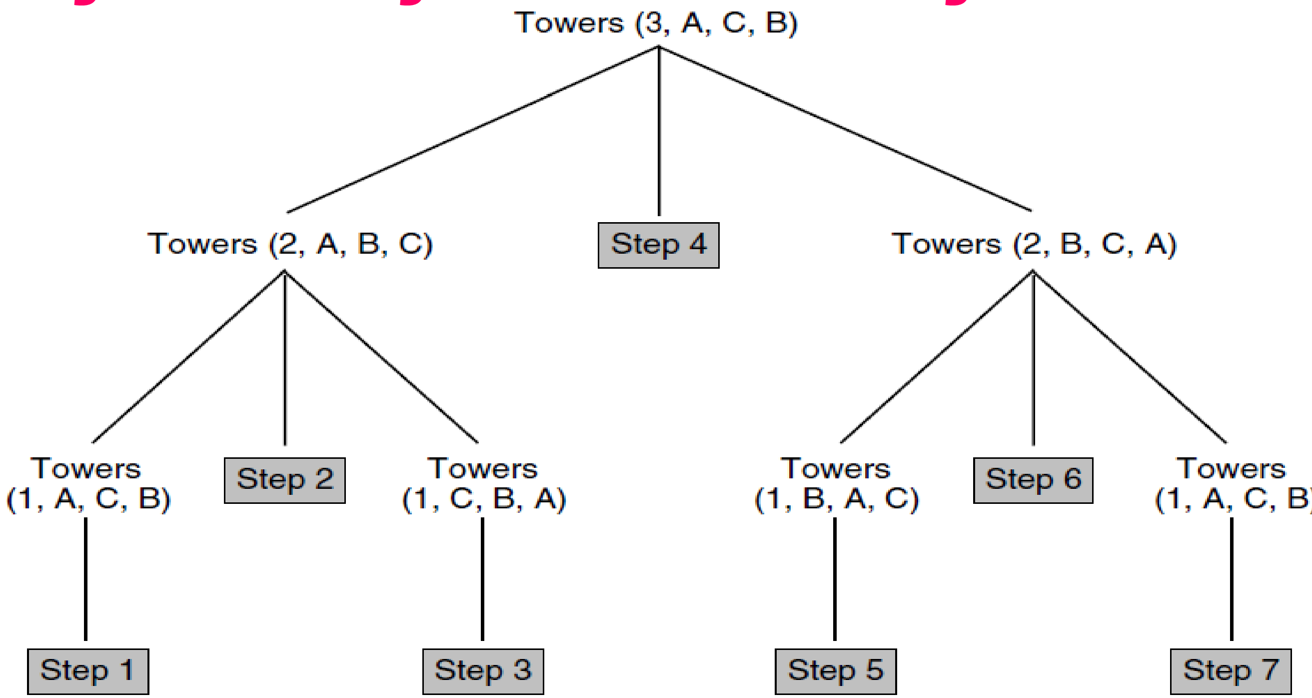
  3 step = step + 1

  4 towers (disks – 1, auxiliary, dest, source, step)

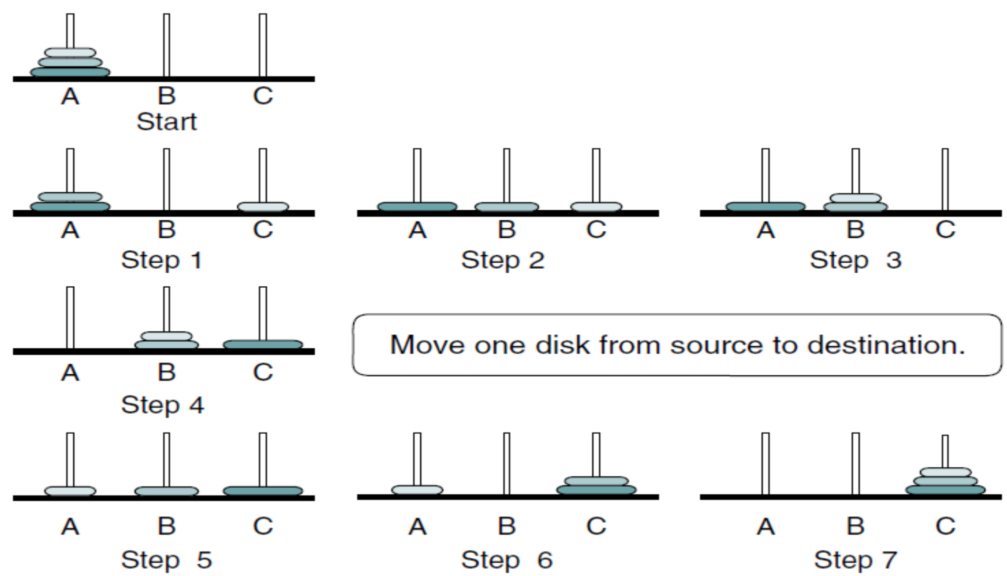
4 return

end towers

Üç disk için Towers çözümü



Towers (3, A, C, B)
Towers (2, A, B, C)
Towers (1, A, C, B)
Step 1 Move from A to C
Step 2 Move from A to B
Towers(1, C, B, A)
Step 3 Move from C to B
Step 4 Move from A to C
Towers(2, B, C, A)
Towers(1, B, A, C)
Step 5 Move from B to A
Step 6 Move from B to C
Towers(1, A, C, B)
Step 7 Move from A to C

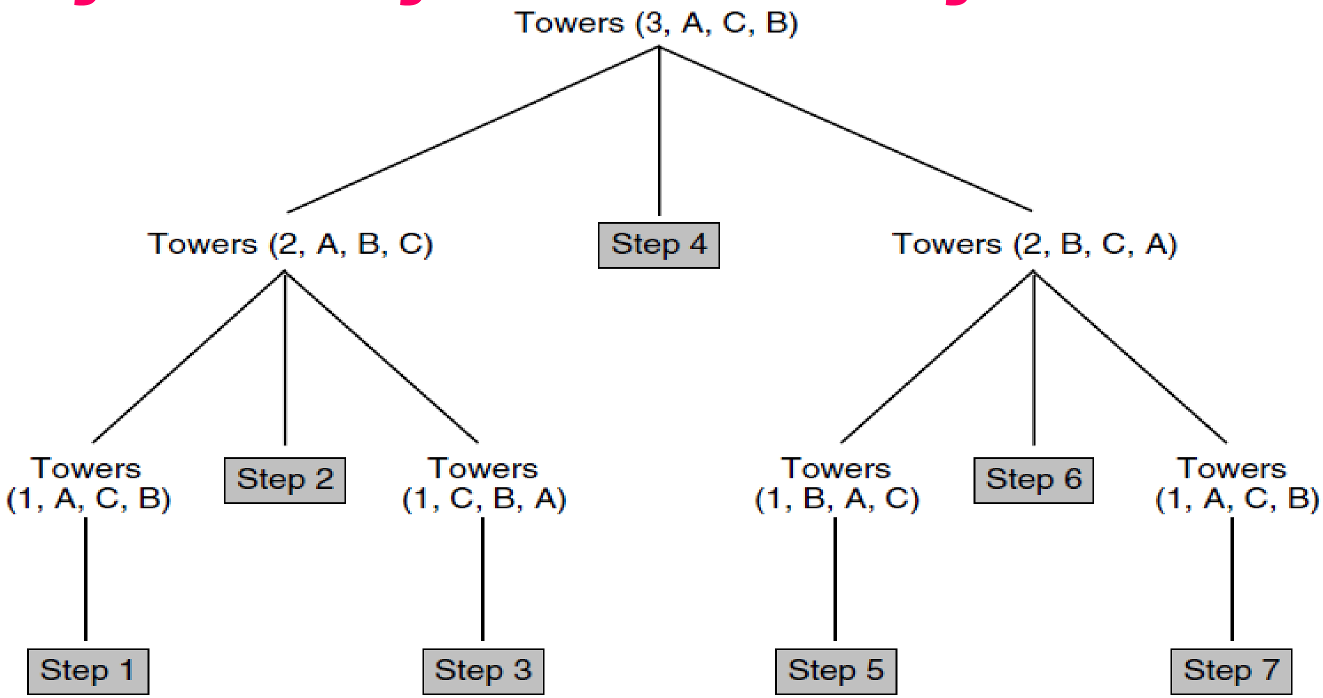


The Towers of Hanoi

ALGORITHM 2-7 Towers of Hanoi

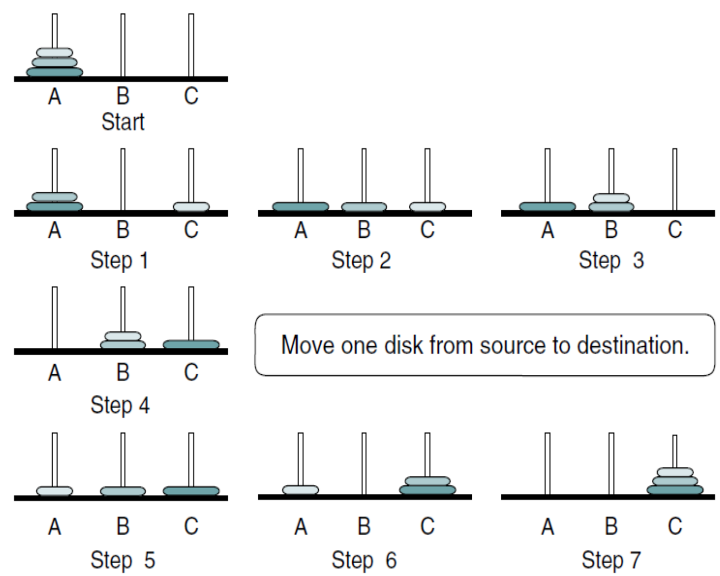
```
Algorithm towers (numDisks, source, dest, auxiliary)
Recursively move disks from source to destination.
   Pre  numDisks is number of disks to be moved
        source, destination, and auxiliary towers given
   Post steps for moves printed
 1 print("Towers: ", numDisks, source, dest, auxiliary)
 2 if (numDisks is 1)
   1  print ("Move from ", source, " to ", dest)
 3 else
   1  towers (numDisks - 1, source, auxiliary, dest, step)
   2  print ("Move from " source " to " dest)
   3  towers (numDisks - 1, auxiliary, dest, source, step)
 4 end if
end towers
```

Üç disk için Towers çözümü



Calls:	Output:	
`Towers (3, A, C, B)`		
`Towers (2, A, B, C)`		
`Towers (1, A, C, B)`		
	`Move from A to C`	Step 1
	`Move from A to B`	Step 2
`Towers (1, C, B, A)`		
	`Move from C to B`	Step 3
	`Move from A to C`	Step 4
`Towers (2, B, C, A)`		
`Towers (1, B, A, C)`		
	`Move from B to A`	Step 5
	`Move from B to C`	Step 6
`Towers (1, A, C, B)`		
	`Move from A to C`	Step 7

FIGURE 2-14 Tracing Algorithm 2-7, Towers of Hanoi



Lab Uygulaması 3

- Ders kitabında PROGRAM 2-4 ile verilen **Towers of Hanoi** C implementasyonunu inceleyip 4 ve 5 diskli durumlarda atılacak adımları takip ediniz.

Alıştırma Sorusu 1

```
algorithm fun1 (x)
1 if (x < 5)
   1  return (3 * x)
2 else
   1  return (2 * fun1 (x - 5) + 7)
3 end if
end fun1
```

- **Yukarıdaki algoritma aşağıda verilen parametrelerle çağırıldığında hangi sonuçları üretir.**

  a. `fun1(4)?` → 3 * 4 = 12

  b. `fun1(10)?` → (2 * (2 * fun1(0) + 7) + 7) = (2 * (2 * (3 * 0) + 7) + 7) = 21

  c. `fun1(12)?` → (2 * (2 * fun1(2) + 7) + 7) = (2 * (2 * (3 * 2) + 7) + 7) = 45

Alıştırma Sorusu 2

```
algorithm fun3 (x, y)
 1 if (x > y)
     1  return -1
 2 elseif (x equal y)
     1  return 1
 3 else
     1  return (x * fun3 (x + 1, y))
 4 end if
end fun3
```

- **Yukarıdaki algoritma aşağıda verilen parametrelerle çağırıldığında hangi sonuçları üretir.**

a. `fun3(10,4)?` → -1

b. `fun3(4,3)?` → -1

c. `fun3(4,7)?` → (4 * fun3(5, 7)) = (4 * (5 * fun3(6, 7))) = 4 * (5 * (6 * fun3 (7, 7))) = 120

d. `fun3(0,0)?` → 1

Alıştırma Sorusu 3

- **Aşağıdaki serinin ilk n elemanını toplayan rekursif algoritmanın sözde kodunu yazınız.**

$$1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\frac{1}{5}+\cdots\frac{1}{n}$$

Çözüm

```
algorithm addN (n)
Bu program yukarıdaki serinin ilk  n elemanını toplar
Pre: n bir integer'dır
Post: float türünde toplam döndürülür.

    if (n == 1)
            return 1
    else
            return (1 / n + addN (n – 1))
    end if
end addN
```

Lab Uygulaması 4:
Yanda sözde kodu verilen algoritmayı C'de kodlayıp çeşitli girişler için çalıştırınız.

Alıştırma Sorusu 4

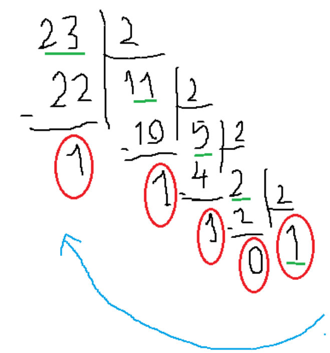
- **İki sayının çarpımını rekursif olarak hesaplama**

Çözüm

```
int recMultiply(int num1, int num2)
{
    if(num1==0 || num2==0) return 0;
    else if (num2<0) return -num1+recMultiply(num1,num2+1);
    else return num1+recMultiply(num1,num2-1);

}   // recMultiply
```

Alıştırma Sorusu 5

- **Bir tamsayıyı ikili sayıya çeviren programı rekürsif algoritmayı C'de kodlayınız.** (Verilen integer sayının binary karşılığını bulup stringe dönüştürünüz.)



$23 \rightarrow (10111)_2$

Çözüm

Rekursif Fonksiyon

```c
void integerToBinary(int num, char* binary)
{
      if (num == 0)  return;  //base case

      integerToBinary(num/2,binary);   //general case
      if(num%2==0) strcat(binary,"0");
      else strcat(binary,"1");

    return;
}
```

Alıştırma Sorusu 5

Çözüm

Ana program

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

//  Prototype Statements
void integerToBinary(int num, char* binary);

int main (void)
{
    char str[50]="";
    int sayi=128;

    integerToBinary(sayi,str);
    printf("\n%d sayisinin binary karsiligi=%s",sayi,str);

    return 0;
}   // main
```