
درس «مبانی کامپیوتر و برنامه‌سازی»

# توابع بازگشتی

صادق علی‌اکبری

# سرفصل مطالب

- مسایلی که به صورت بازگشتی حل می‌شوند
- تابع بازگشتی
- کاربردهای توابع بازگشتی
  - چند مثال

# راه‌حل‌های بازگشتی

- برخی از مسایل، ذاتاً بازگشتی تعریف می‌شوند
  - مثلاً: فیبوناچی n اُم عبارت است از مجموع فیبوناچی n-1 اُم و فیبوناچی n-2 اُم
- راه‌حل برخی از مسایل به صورت **بازگشتی** قابل بیان است
  - محاسبه n!
    - راه حل بازگشتی: $n! = n * (n-1)!$
    - راه حل غیربازگشتی : $n! = 1 * 2 * .... * n$
- در راه‌حل‌های بازگشتی، شرایط پایه را باید مشخص کنیم. مثلاً:
  - فیبوناچی یکم و دوم برابر با ۱ هستند.
  - یک‌فاکتوریل (1!) برابر با ۱ است.
- در این صورت راه‌حل به صورت استقرایی قابل اجرا است (چرخه بینهایت تشکیل نمی‌شود)

# توابع بازگشتی

- می‌دانیم هر تابع می‌تواند توابع دیگری را فراخوانی کند
  - فراخوانی تودرتو
  - مثلاً در پیاده‌سازی permutation و combination از تابع factorial استفاده کردیم
- اما هر تابع می‌تواند خودش را هم فراخوانی کند
  - مثل یک تابع دیگر، اما خودش را صدا می‌زند
  - همه فرایند، مثل فراخوانی یک تابع دیگر است (مراسم صدا زدن، پشته، ارسال پارامترها و ...)
- به این‌گونه توابع (که خودشان را فراخوانی می‌کنند) **تابع بازگشتی** می‌گویند

# مثال: فاکتوریل

- تابع فاکتوریل را به صورت بازگشتی پیاده‌سازی کنید

```
int factorial(int n){
   if(n==1) return 1;
   return n* factorial (n-1);
}
```

- به شرایط پایه دقت کنید
  - n==1 ← n! = 1
- بدون شرایط پایه، چه مشکلی وجود داشت؟
- کدام‌یک ساده‌تر است؟

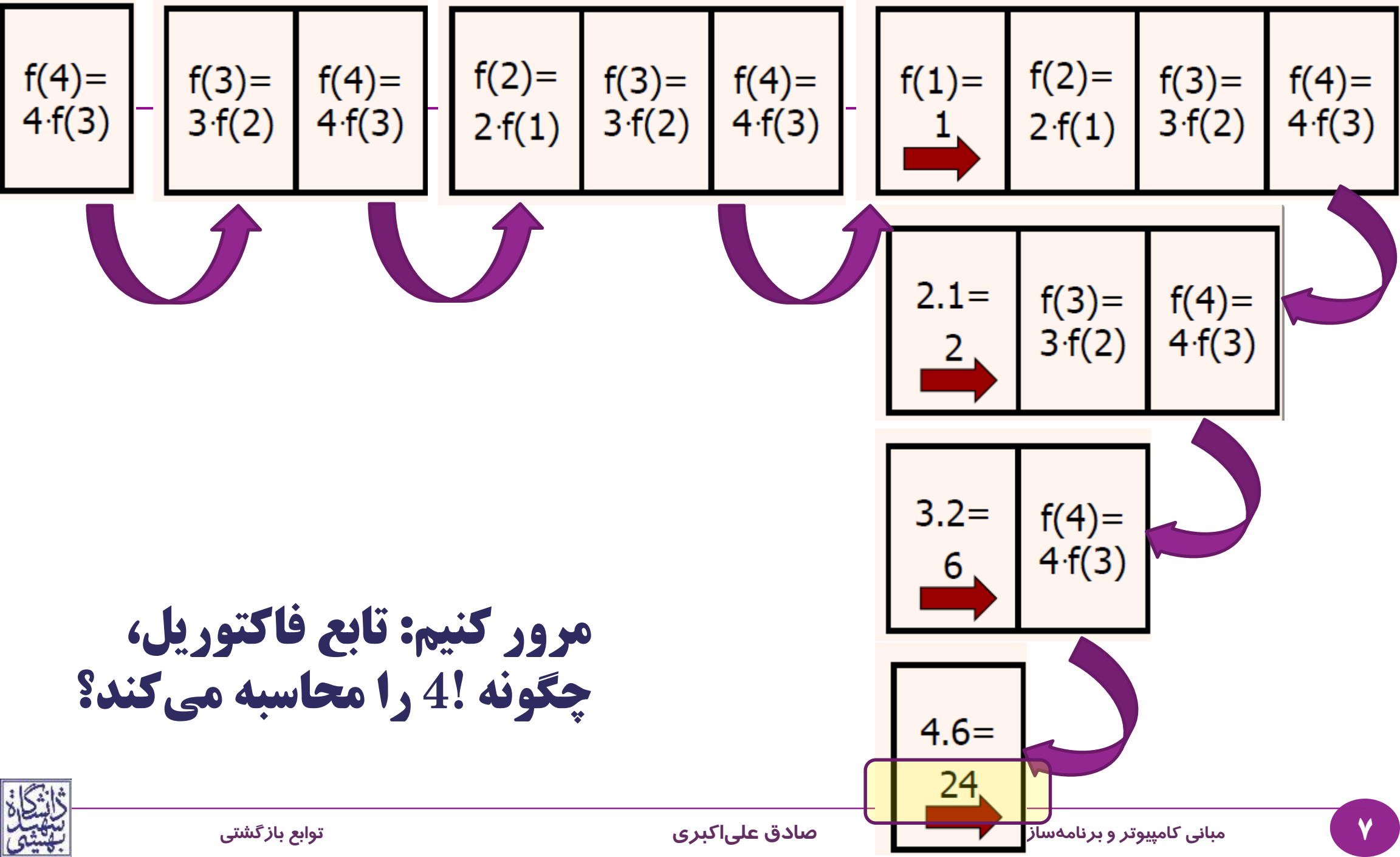
```
int factorial(int n) {
   if(n<0)
      return 0;
   int f = 1;
   while(n>1)
      f*=n--;
   return f;
}
```

# بررسی پشته فراخوانی تابع factorial

```
int factorial(int n){
      if(n==1) return 1;
      int result=n*factorial(n-1);
      return result;
}
```

- فرض کنید factorial(3) فراخوانی شود

| | |
|---|---|
| result | |
| n | 1 |
| result | ? |
| n | 2 |
| result | ? |
| n | 3 |




**مرور کنیم: تابع فاکتوریل،**
**چگونه !4 را محاسبه می‌کند؟**

# تمرین

```
int fib(int n);
```

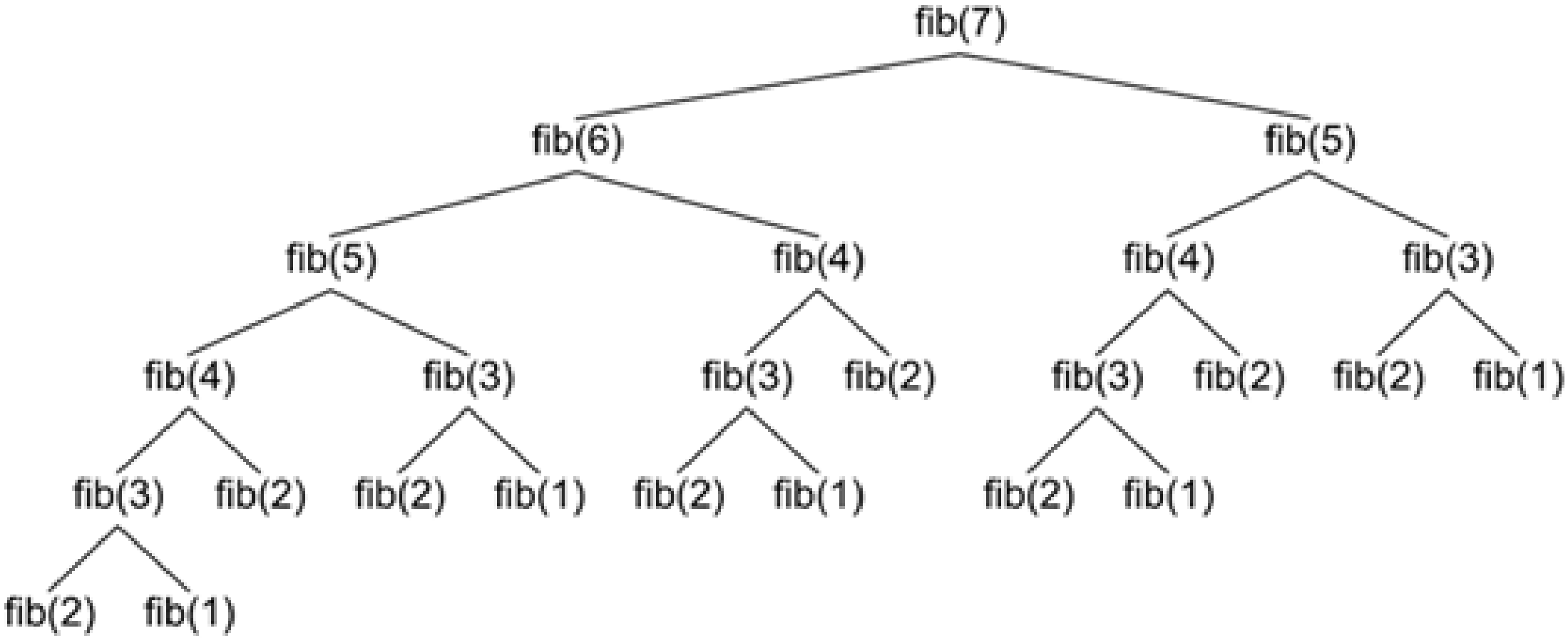
- تابع فیبوناچی را به صورت بازگشتی پیاده‌سازی کنید
- به شرایط پایه دقت کنید
- کدام یک ساده‌تر است؟
- کدام‌یک سریع‌تر است؟

```
int fib(int n) {
    if(n<=0)
        return 0;
    if(n==1)
        return 1;
    int a = 1, b = 1;
    for (int i = 3; i <= n; i++) {
        int c = a + b;
        a = b;
        b = c;
    }
    return b;
}
```

```
int fib(int n){
  return (n==1 || n==2) ? 1 : fib(n-1)+fib(n-2);
}
```

# درخت اجرای fib(7)

```
int fib(int n){
  return (n==1||n==2)?1:fib(n-1)+fib(n-2);
}
```



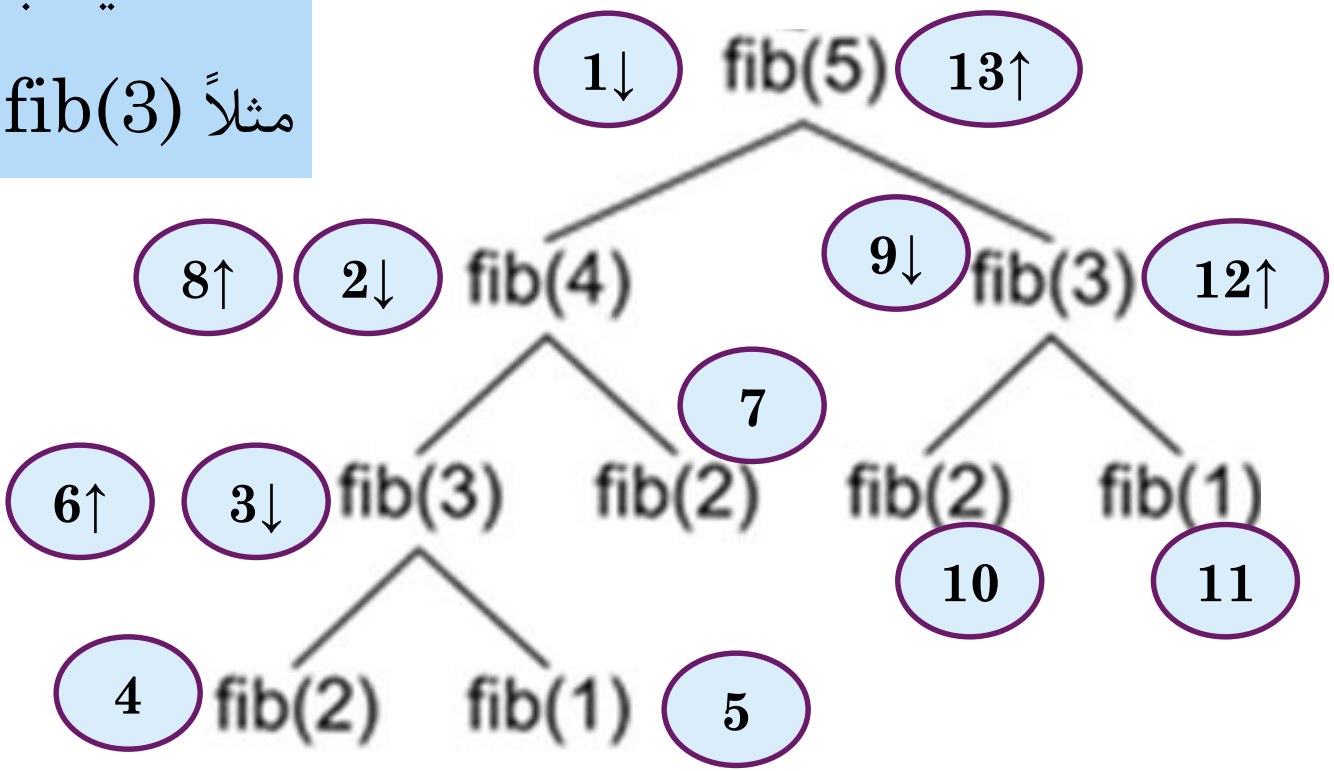
# تمرین

```
int fib(int n){
  return (n==1 || n==2) ? 1 : fib(n-1)+fib(n-2);
}
```

- فراخوانی تابع fib(5) را دنبال کنید

دقت کنید: بعضی از زیرمسأله‌ها چند بار حل شدند

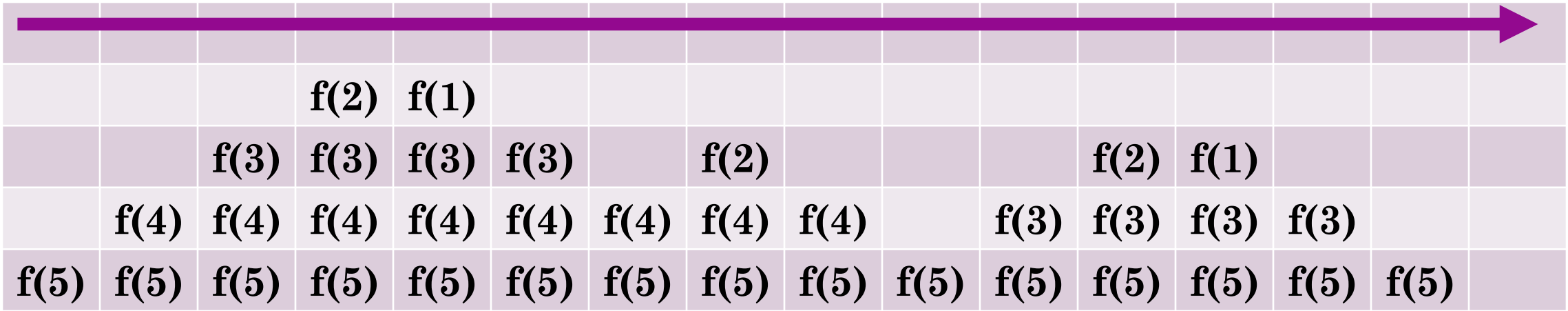
مثلاً fib(3) دو بار محاسبه شد



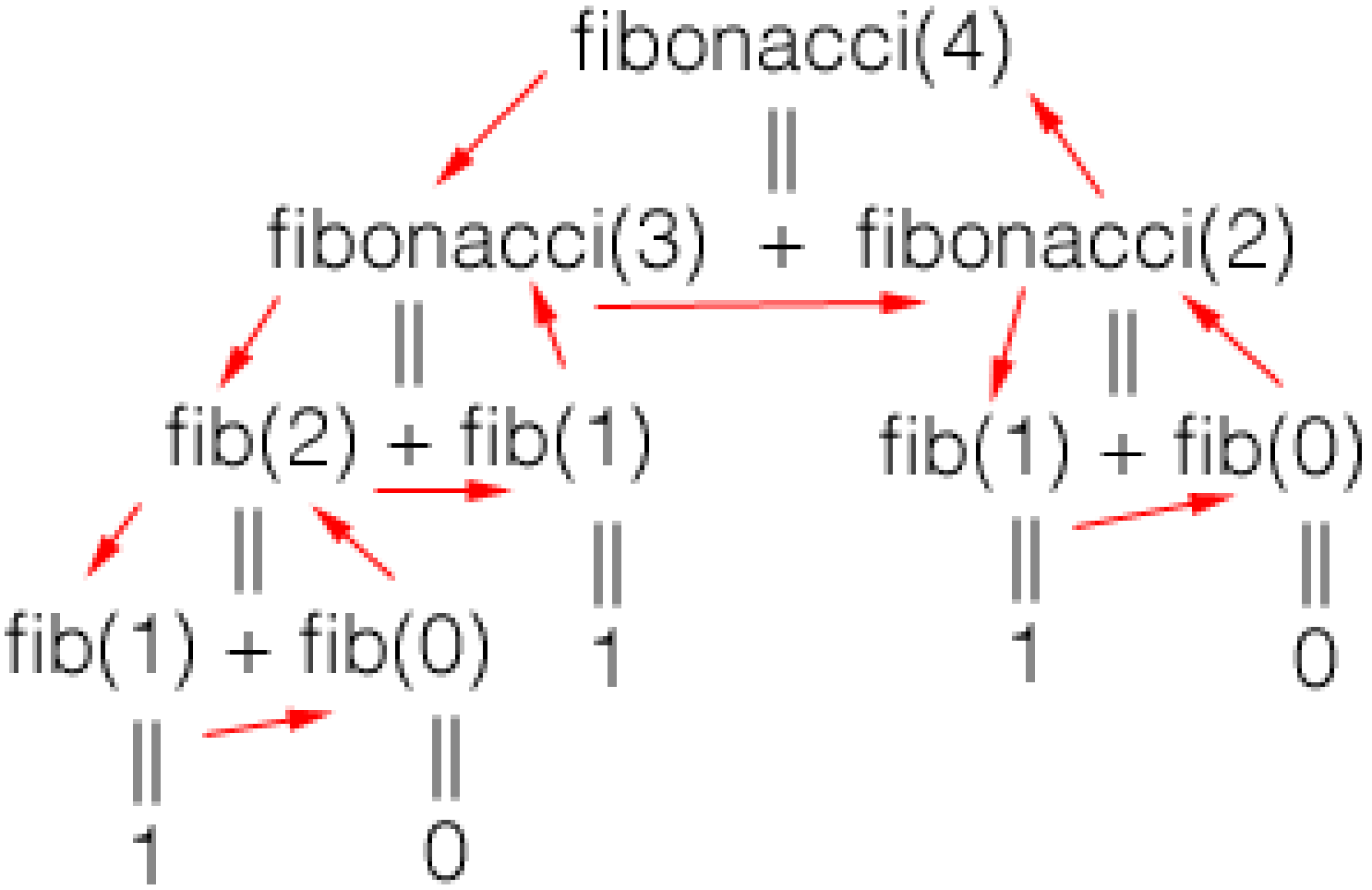
# تمرین

```
int fib(int n){
  return (n==1 || n==2) ? 1 : fib(n-1)+fib(n-2);
}
```

- پشته فراخوانی fib(5)



# جریان اجرای fib(4)



# مرور راه‌حل‌های بازگشتی

- در این رویکرد، مسأله را به مسایل کوچکتر تقسیم می‌کنیم
  - اگر مسأله کوچکتر حل شود، حل مسأله بزرگتر آسان است
- مثال: fib(n) را با کمک fib(n-1) و fib(n-2) حل می‌کنیم
- مثال: factorial(n) را با کمک factorial(n-1) حل می‌کنیم
- همچنین یک شرایط اولیه، برای خاتمه شرایط بازگشتی تعیین می‌کنیم
  - تا ابد که نمی‌شود مسأله را به مسایل کوچکتر تقسیم کرد!
  - بالاخره در کوچکترین مسأله، جواب بدیهی یا ساده وجود دارد
  - مثلاً: 1 = !1 یا fib(1)=fib(2)=1

# مثال: برنامه زیر چه چیزی نمایش می‌دهد؟

```
void show_number(int n)
{
   cout<<n<<endl;
   show_number(n-1);
}
int main()
{
   show_number(5);
}
```

```
5
4
3
2
1
0
-1
-2
...
```

- و بعد از چاپ تعداد زیادی عدد، برنامه با خطا مواجه می‌شود و اجرایش قطع می‌شود

# مثال: برنامه زیر چه چیزی نمایش می‌دهد؟

```
void show_number(int n)
{
   if(n==0)
      return;
   cout<<n<<endl;
   show_number(n-1);
}
int main()
{
   show_number(5);
}
```

```
5
4
3
2
1
```

```
void show_number(int n)
{
   if(n==0)
      return;
   cout<<n<<endl;
   show_number(n-1);
   cout<<n<<endl;
}
int main()
{
   show_number(5);
}
```

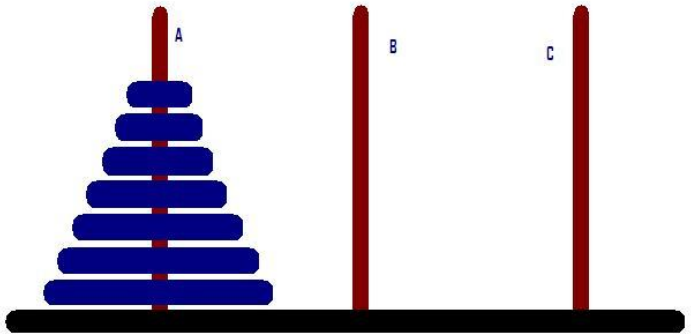
```
5
4
3
2
1
1
2
3
4
5
```

# حلقه یا بازگشتی؟ مسأله این است...

بسیاری از مسایل را هم با حلقه و هم به صورت بازگشتی می‌توانیم حل کنیم

- مزایا و معایب راه‌حل بازگشتی:
- عیب: معمولاً کارایی کمتری دارد (از نظر سرعت و حافظه بدتر است)
  - بار بالا به خاطر فراخوانی تودرتو و بازگشتی تابع
  - نگه داشتن آدرس بازگشتی، مراسم فراخوانی تابع، ایجاد متغیرها، پشته بزرگ می‌شود و ...
- عیب: ممکن است یک زیرمسأله چند بار حل می‌شود
  - مثلاً در جریان محاسبه حل $fib(4)$ ، دو بار $fib(2)$ محاسبه می‌شود
- مزیت: حل برخی مسایل به صورت غیربازگشتی، بسیار سخت‌تر یا گاهی غیرممکن است
- مزیت: معمولاً برنامه ساده‌تر و کوتاه‌تر است
  - خوانایی و فهم برنامه راحت‌تر است
  - معمولاً طراحی راه‌حل ساده‌تر است

# مسأله برج هانوی




- در معبدی در شرق آسیا، سه میله الماسی قرار داشت که یکی از آنها حاوی ۶۴ قرص (حلقه) طلا بود
- حلقه‌ها در اندازه‌های مختلف به ترتیب نزولی روی هم چیده شده بودند
- کاهنان معبد می‌خواستند حلقه‌های طلا را از آن میله به میله‌ای دیگر انتقال دهند
- به نحوی که هیچ‌گاه یک حلقه بزرگ‌تر روی یک حلقه کوچک‌تر قرار نگیرد
- آن‌ها باور داشتند که با تمام شدن انتقال حلقه‌ها، عمر جهان نیز به پایان خواهد رسید

# مشاهده

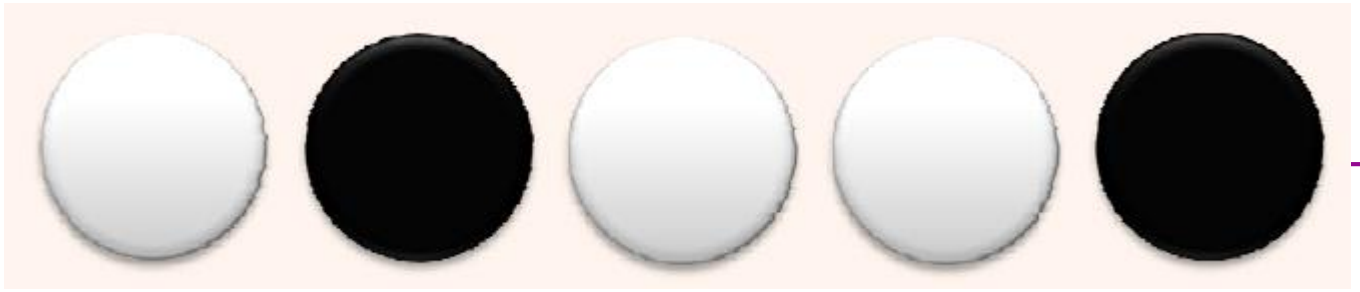
# راه حل؟

```
void Hanoi(int m, char from, char help, char to);
int main() {
   int discs;
   cout << "Enter the number of discs: " << endl;
   cin >> discs;
   Hanoi(discs, 'A', 'B', 'C');
}
void Hanoi(int m, char from, char help, char to) {
   if (m == 1)
      cout << from << " => " << to << endl;
   else {
      Hanoi(m - 1, from, to, help);
      cout << from << " => " << to << endl;
      Hanoi(m - 1, help, from, to);
   }
}
```

```
Enter the number of discs:
3
A => C
A => B
C => B
A => C
B => A
B => C
A => C
```

# تمرین

- آیا کاهنان درست فکر می‌کردند؟
- با پایان کار انتقال حلقه‌ها، عمر دنیا به پایان می‌رسد؟
- چند حرکت برای انتقال ۶۴ حلقه لازم است؟
- فرض کنید هر حرکت یک ثانیه طول می‌کشد
- نتیجه: ۵۸۵ میلیارد سال
- برنامه‌ای بنویسید که تعداد حرکت‌های لازم را حساب کند
  - کافیست برنامه‌ای که برای حل مسأله برج هانوی نوشتید را اندکی تغییر دهید
  - راه حل بهتری برای پیدا کردن تعداد حرکت‌های لازم؟
  - ثابت کنید تعداد حرکت‌های لازم $2^N-1$ است ($N$ تعداد حلقه‌هاست)
- اگر مسأله فقط پیدا کردن تعداد حرکت‌های لازم بود، آیا راه حل بازگشتی مناسب بود؟



# تمرین

- فرض کنید N سکه در اختیار دارید که هر یک داری دو طرف سفید و سیاه هستند
- با این فرض که هیچ دو سکه سیاهی کنار هم نباشند،
- به چند حالت می‌توانیم این سکه‌ها را کنار هم قرار دهیم؟
- تابعی بازگشتی بنویسید که N را به عنوان پارامتر بگیرد و تعداد حالت‌های ممکن را برگرداند
- راهنمایی:

$F(1) = 2$

$F(2) = 3$

# پاسخ

- فرض کنید تعداد حالت‌های ممکن برای چینش N سکه، F(N) باشد
- پاسخ را به دو زیرمسأله تقسیم می‌کنیم:

۱- در خانه اول سکه سفید بگذاریم و ۲- در خانه اول سکه سیاه بگذاریم

- اگر در خانه اول، سکه سفید بگذارید:
  - N-1 خانه بعدی را هرطور بخواهید می‌توانید بچینید ← F(N-1)
- اگر در خانه اول، سکه سیاه بگذارید:
  - مجبورید خانه دوم را سفید بگذارید
  - و آن‌گاه N-2 خانه بعدی را هرطور بخواهید می‌توانید بچینید ← F(N-2)

**F(N) = F(N-1)+F(N-2)**
**F(1) = 2**
**F(2) = 3**

# جمع‌بندی

# جمع‌بندی

- حل مسایل به صورت بازگشتی
- توابع بازگشتی
- مزایا و معایب توابع بازگشتی

# مطالعه

- Chapter 5 of C How to Program (Deitel & Deitel), 7th edition
  - و یا بخش‌های متناظر در کتاب C++

# جستجوی بیشتر

- مفهوم برنامه‌نویسی پویا به چه معناست و چه ارتباطی با برنامه‌نویسی بازگشتی دارد؟ (Dynamic Programming)

# پایان