تبدیل الگوریتم های بازگشتی به غیر بازگشتی

مفهوم recursion یک ایده ی بنیادی در علم کامپیوتر است. روشی است که در آن یک مسأله بزرگ با تقسیم به زیر مسایلی از نوع خودش حل میشود. بنابراین حل مسأله به روش بازگشتی درواقع یک روش divide and conquer است که مسأله پیچیده را به مسایل کوچک و قابل حل تبدیل میکند. برخی مسایل مثل برج هانوی ماهیتا بازگشتی هستند اما توابع بازگشتی ازنظر عملکرد ضعیف هستند و صورت غیربازگشتی آنها بهینه تر پیاده سازی میشود .از اینها گذشته برخی زبان های برنامه نویسی از صورت بازگشتی حمایت نمیکنند (زبان های بدون پشته) به همین دلیل تبدیل به صورت غیربازگشتی اهمیت میابد. دو روش برای این تبدیل وجود دارد:

1-روش تبدیل به حلقه تکرار برای مسایل tail recursive 2- استفاده از پشته(stack)

1- Tail Recursive :

توابعی را tail recursive گویند که حداکثر یک فراخوانی بازگشتی داشته باشند و بلافاصله بعداز آن فراخوانی ,کار تابع تمام شود (یعنی بعداز بازگشت هیچ عملیاتی روی مقادیر بازگشتی تابع انجام نشود).

این توابع را بدون رویه ی پیچیده میتوان با استفاده از حلقه تکرار به غیربازگشتی تبدیل کرد. چون در پایان توابع tail recursive تغییری بر مقدار بازگشتی توابع رخ نمیدهد و فقط بازگشت انجام میشود بنابراین نیازی به پشته برای ذخیره سازی اطلاعات نیست. بنابراین هم ازنظر حافظه ی مصرفی هم هزینه ی زمانی بهینه هستند.

مثال زیر تابع جستجوی دودویی را که به صورت بازگشتی پیاده سازی میشود به غیربازگشتی تبدیل کرده.

صورت بازگشتی:

```
BinarySearch(v[],low, high, const value)
    If (low<high)
        Mid=(low+high)/2
        If(v[mid]<value)
            Return BinarySearch(v,mid+1,high,value)
```

```
        Else
            Return BinarySearch(v,low,mid,value)
    Else
        Return low
```

تبدیل شده به صورت غیربازگشتی با استفاده از حلقه تکرار :

```
BinarySearch(v[],low, high, const value)
    Low=0
    High=n
    While(low<high)
        Mid=(low+high)/2
        If(v[mid]<value)
            Low=mid+1
        Else
            High =mid
    Return low
```

عمق پشته در "QuickSort"

الگوریتم QuickSort شامل دو فراخوانی بازگشتی از خودش است. بعد از فراخوانی تابع Partition به صورت بازگشتی قسمت چپ وسپس راست آرایه را مرتب می کند.

```
QuickSort(A,p,r)
  if(p<r)
    q=Partition(A,p,r)
    QuickSort(A,p,q-1)
```

QuickSort(A,q,r)

در واقع لزومی به فراخوانی دومین تابع بازگشتی نیست و می توان آن را با ساختار کنترلی تکراری پیاده سازی کرد. یعنی روش tail recursion که توسط برخی کامپایلر ها استفاده می شود.

کد زیر پیاده سازی الگوریتم QuickSort به روش tail recursion است :

```
TAIL_RECURSIVE_QUICKSORT(A,p,r)
    while p<r
      // partition and sort left subarray
      q=PARTITION(A,p,r)
      TAIL_RECURSIVE_QUICKSORT(A,p,q-1)
      p=q+1
```

کامپایلر ها معمولا از پشته (stack) برای پیاده سازی فرایند بازگشتی استفاده می کنند که شامل اطلاعاتی ازجمله ارزش پارامترها برای هر فراخوانی بازگشتی است.اطلاعات فراخوانی اخیر در بالای پشته و اطلاعات فراخوانی های پیشین در پایین پشته قرار دارند. طی فراخوانی یک بازگشت اطلاعات ان فراخوانی در پشته قرار می گیرد (push میشود) و با اتمام آن اطلاعاتش از پشته خارج میشود (pop میشود). اگر فرض کنیم آرایه ی پارامتر های یک فراخوانی به وسیله ی پوینتر مشخص شود , در اینصورت اطلاعات هر فراخوانی روی پشته به O(1) فضای پشته نیاز دارد در اینصورت عمق پشته بیشترین فضای مورد استفاده از پشته است.

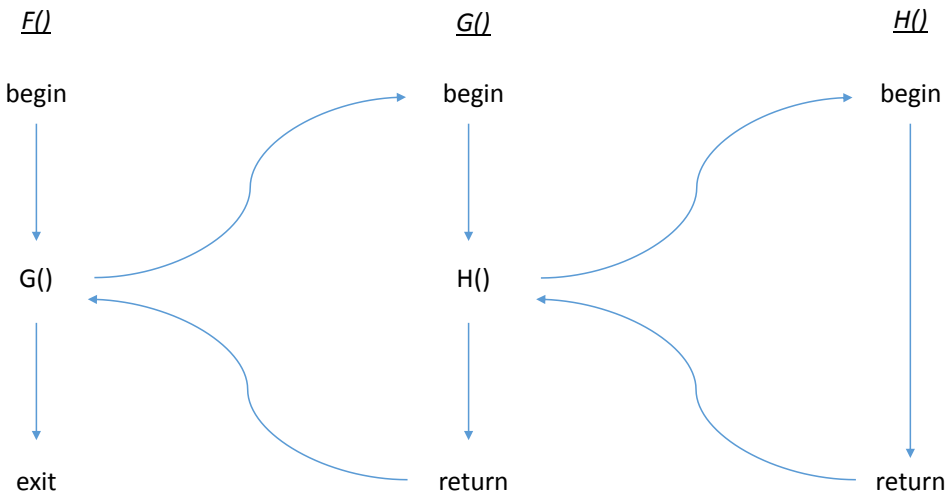
2-تبدیل الگوریتم های بازگشتی به غیر بازگشتی با استفاده از پشته :

هر تابع بازگشتی را با استفاده از پشته میتوان به روشی که شرح داده خواهد شد به تابع غیر بازگشتی تبدیل کرد.یکی کاربرد های پشته فراخوان های بازگشتی می باشد که در تبدیل آن ها به غیر بازگشتی هم کاربرد دارند. هدف این بخش شناختن بیشتر رفتار و مدل الگوریتم های بازگشتی و تبدیل آن ها به مدل ساده تر غیربازگشتی است تا بتوان در جاهایی که امکان استفاده از الگوریتم های بازگشتی نیست (مثلا زبان هایی که فاقد پشته هستند) استفاده کرد.

ابتدا لازم است توضیحی کوتاه در مورد فراخوانی بدهیم.

فراخوانی دو بخش است: ابتدا عمل فراخوانی و سپس بازگشت از یک فراخوانی.

در شکل1 مثالی از فراخوانی آمده که نشان می دهد در آن تابع F() در برنامه خود تابع G() را و تابع G() تابع H() را فراخوانی می کند. تابع H() در پایان کار خود به تابع G() بر می گردد و تا انتهای آن دستورات را انجام داده و پس از اتمام کار های G() به تابع F() بر می گردد و تا پایان آن دستورات انجام و با اتمام کار های A خارج می شود.



مثال فراخوانی و بازگشت

فراخوانی وقفه ای در سخت افزار ایجاد می کند و در آن لحظه تمام مقادیر و متغیر ها ذخیره می شوند و بعد از اعمال فراخوانی با بازگشت وضعیت قبل فراخوانی شده و مجددا بازسازی می شود.

نکات مورد توجه در فراخوانی توابع:

- فراخوانی بازگشت در یک تابع همانند فراخوانی تابعی دیگر در آن است.
- فراخوانی هر تابعی درون تابع دیگر نیازمند حافظه ای برای ذخیره متغییر های محلی و پارامترهاست.
- قبل از فراخوانی تابع دیگر باید آدرس بازگشت ذخیره شود.
- برای حافظه ی مورد نیاز میتوان از پشته استفاده کرد.

- بعد از اتمام فراخوانی باید به آدرس بازگشت که قبلا ذخیره شده بازگردیم و در صورت نیاز متغییر های محلی و پارامتر ها را به روز کنیم.

مراحلی که در هر فراخوانی(call) انجام می شود:

مقدار اولیه دهی:

- پشته ای برای ذخیره متغییر های محلی و پارامترها و آدرس های بازگشت
- اولین جمله بعداز مقدار دهی پشته برچسب میخورد (برای ذخیره محل شروع کار پشته)

مراحل جایگزینی:

- ذخیره همه ی متغییر های محلی و پارامتر ها در پشته
- قرار دادن یک عدد مثل i در پشته برای نگهداری آدرس بازگشت (return address)
- Set کردن مقدار پارامترها (parameter passing)
- ذخیره PC در پشته (program counter)
- همیشه جمله ی بلافاصله بعداز فراخوانی بازگشت برچسب میخورد (چون آدرس بازگشت را مشخص میکند) (return value)

مراحلی که هنگام بازگشت (return) انجام می شود:

متغیر های محلی از پشته pop می شوند؛

آدرس بازگشت از پشته pop می شود؛

آخرین رکورد از پشته pop می شود(PC)؛

ادامه کار از آدرس بازگشت.

نکته: Recursive و Non-Recursive از دید کامپایلر فرق ندارند.

سوال:آیا همیشه میتوان یک مسأله بازگشتی را به غیر بازگشتی تبدیل کرد؟

باتوجه به نکات فوق و اینکه میتوان مستقیما به پشته دسترسی داشت , میتوان هر مسأله بازگشتی را به مسأله غیر بازگشتی تبدیل کرد.و به جای انکه پارامتر ها و توابع در فرایند بازگشت در پشته جای گیرند به وسیله ی حلقه تکرار اینکار انجام شود.

ولی ممکن است این تبدیل ساده نباشد. دلیل این است که برای حل یک مساله که ماهیتاً بازگشتی است باید به طور بازگشتی به مسأله فکر کرد ولی از آنجا که ذهن انسان یک پشته ای می باشد این کار برای انسان مشکل است.

توابع بازگشتی و غیربازگشتی هردو قدرتمند هستند. هر تابع بازگشتی را میتوان به توابع غیربازگشتی تبدیل کرد و بالعکس. اینکه کدام روش مناسب تر است به مسأله بستگی دارد.

- آسانی بیان : برخی مسایل به طور بازگشتی به سادگی بیان میشوند و برخی به صورت غیر بازگشتی
- عملکرد : معمولا (نه همیشه) صورت غیربازگشتی سریعتر از بازگشتی اجرا میشود

فاکتوریل:

پیاده سازی تابع فاکتوریل اعداد طبیعی به صورت بازگشتی :

```
def Factorial (n):
    if n<=1:
        return 1
    else:
        return (n*Factorial(n-1))
```

به صورت غیر بازگشتی:

```
def Factorial(n):
    result=1
    for(i=2;i<n+1;i++)
        result*=I
    return result
```

همانطور که می بینیم تابع بازگشتی فاکتوریل tail recursive نیست چون بعد از بازگشت باید عمل ضرب انجام شود.

بررسی بازگشتی 4! :

$b_4 = 4 * b_3$

$= 4 * (3 * b_2)$

$= 4 * (3 * (2 * b_1))$

$= 4 * (3 * (2 * (1 * b_0)))$

$= 4 * (3 * (2 * (1 * 1)))$

$= 4 * (3 * (2 * 1))$

$= 4 * (3 * 2)$

$= 4 * 6$

$= 24$

N	Recursive	Iterative
10	334 ticks	11 ticks
100	846 ticks	23 ticks
1000	3368 ticks	110 ticks
10000	9990 ticks	975 ticks
100000	stack overflow	9767 ticks

در بررسی زمانی تابع فاکتوریل به صورت بازگشتی و غیربازگشتی می بینیم غیر بازگشتی سریعتر است.

ب.م.م:

الگوریتم اقلیدسی برای محاسبه ی بزرگترین مقسوم علیه مشترک (ب.م.م : gcd) را می توان به صورت بازگشتی نوشت:

$$\gcd(x,y) = \begin{cases} x & \text{if } y = 0 \\ \gcd(y, \text{remainder}(x,y)) & \text{if } y > 0 \end{cases}$$

در روابط زیر x%y بیانگر باقیمانده ی x/y است:

$$\gcd(x,y) = \gcd(y, x\%y) \text{ if } y \neq 0$$

$$\gcd(x,0) = x$$

محاسبه ی ب.م.م 27 و9 :

gcd(27,9) = gcd (9, 27%9)

= gcd (9,0)

= 9

محاسبه ی ب.م.م 111 و 259 :

gcd (111,259) = gcd (111,259%111) = gcd (111,37)

= gcd (37,111%37) = gcd (37,0)

= 37

کد بازگشتی فوق tail recursion است و می توان آن را به کدی با ساختار کنترلی تکراری تبدیل کرد.

```
gcd (x , y)
    while (y !=0)
        reminder =x%y
        x=y
        y= reminder
    return x
```

برج هانوی:

$$\text{hanoi}(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2 \cdot \text{hanoi}(n-1) + 1 & \text{if } n > 1 \end{cases}$$

محاسبه ی *Hanoi*(4) :

Hanoi (4) = 2* Hanoi (3) + 1

= 2*(2* Hanoi (2) + 1) + 1

= 2*(2*(2* Hanoi (1) + 1) + 1) + 1

=2*(2*(2*1 + 1) + 1) + 1 = 2*(2*(3) + 1) + 1= 2*(7) + 1 = 15

یافتن فرمولی روشن برای هانوی:

H_1= 1 = 2^1 -1

H_2= 3 = 2^2 -1

H_3= 7 = 2^3 -1

H_4= 15 = 2^4 -1

H_5= 31 = 2^5 -1

H_6= 63 = 2^6 -1

H_7= 127= 2^7 -1

for all n>=1 : H_n = 2^n -1

قسمت اول(تحلیل بازگشتی):

RecHanoi(n,f,t,h)

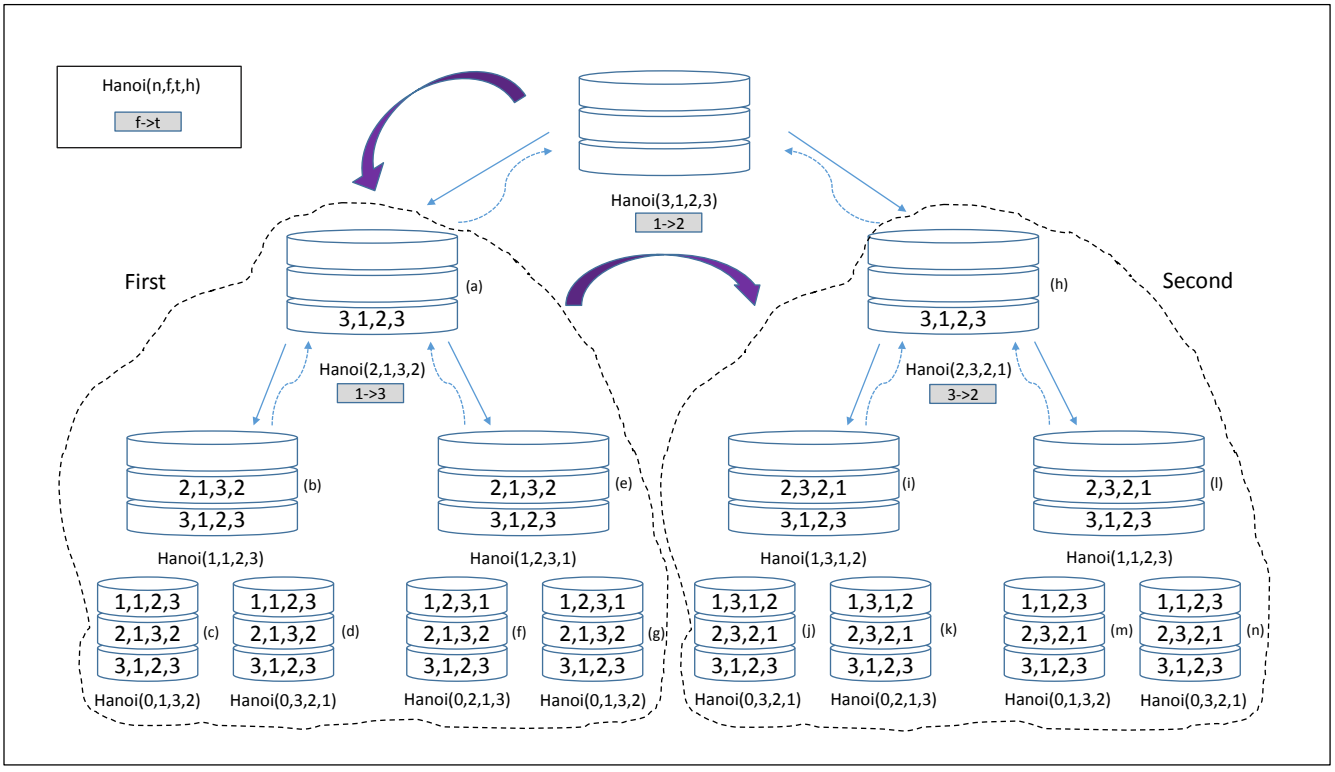
1. If(n=0)

2. return

3. RecHanoi(n-1,f,h,t)

4. A: Write(f->t)

5. B: RecHanoi(n-1,h,t,f)



مراحل انجام Hanoi با استفاده از پشته

درپیاده سازی برج هانوی با استفاده از پشته ، در ابتدا پشته خالی است و با اولین فراخوانی تابع Hanoi(3,1,2,3) در ابتدا مقادیر t،f،n و h و سپس آدرس بازگشت در پشته قرار می گیرند. در این حالت n=3، f=1، t=2 و h=3 می باشد و آدرس بازگشت را A در نظر می گیریم که در تابع Hanoi به عنوان آدرس 3 نشان داده شده است. سپس انتقال پارامتر انجام می شود، یعنی n=2 و f همان مقدار خود را حفظ و t و h با هم جابه جا می شوندو دوباره کار الگوریتم از ابتدا آغاز می شود. (توضیح حالت (a) که Hanoi(2,1,3,2) فراخوانی شده است)

در حالت کلی در هر مرحله ابتدا تمام زیر مجموعه سمت چپ هر پشته و سپس سمت راست آن اجرا می شود. در هر مرحله پشته مورد نظر در سمت چپ خود Hanoi(n-1,f,h,t) و در سمت راست خود Hanoi(n-1,h,t,f) را فراخوانی می کند. این کار تا جایی ادامه پیدا می کند که n=0 شود.در هر فراخوانی تابع هانوی هم بعد بازگشت از اولین فراخوانی درون آن دستور write اجرا می شود و f->t را در خروجی می نویسد.

دنباله پیشروی در پشته های مثال مذکور هم به ترتیب زیر است:

a,b,c,d,e,f,g,h,I,j,k,l,m,n

قسمت دوم(تحلیل غیر بازگشتی):

```
NonRecHanoi(n,f,t,h)
1. STACK s
2. Char addr
3. START:
4. If(n==0)
5.       goto RET
6. s.push('A')
7. s.push(n,f,t,h)
8. n--
9. swap(h,t)
```

```
10. goto START
11. RET:
12. If(s.isEmpty())
13.     return;
14. s.pop(n,f,t,h,addr)
15. if(addr==’A’)
16.     write(f->t)
17.     s.push(n,f,t,h,’B’)
18.     n--
19.     swap(h,f)
20.     goto START
21. else
//addr == ‘B’
22.     goto RET
```

نکته: کاراکتر های A و B را به عنوان آدرس بازگشت در نظر گرفتیم چون آدرس واقعی آن ها در زمان اجرا مشخص می شود.

در تابع بالا پیاده سازی ای غیر بازگشتی از تابع هانوی ارائه شده است که ابتدا پشته s را که تهی است ایجاد می کنیم.هر رکورد این پشته شامل مقادیر n، f، t و h و آدرس های بازگشت ’A‘ یا ’B‘ می باشد. برچسب START در تابع به معنی شروع فراخوانی بازگشتی است که با دستور goto START آغاز می شود. برچسب RET هم برای عمل بازگشت است که با goto RET شروع می شود. مانند الگوریتم بازگشتی، هر فراخوانی از آغاز برچسب START (خط 3) شروع می شود و اگر n=0 باشد تنها مهره بالایی را حرکت داده و بازمی گردد و در غیر این صورت مقادیر فعلی متغیر ها و آدرس بازگشت را بالای پشته وارد می کند، پس از انتقال پارامتر ها

برای انجام فراخوانی بازگشتی بعدی به START رفته و دوباره مراحل گفته شده را طی می کند.عمل بازگشت هم که از برچسب RET (خط 11) شروع می شود به این ترتیب است که اگر پشته تهی بود یعنی آخرین بازگشت است و باید از تابع مذکور خارج شده و به تابعی که آن را فراخوانده بازگردد. اگر پشته تهی نباشد باید مقادیر بالای آن را دریافت و پس از مقدار دهی به متغیر ها دور ریخته شوند و با توجه به آدرس بازگشت عمل مناسب را انجام دهد. آدرس بازگشت دو مدل رفتار می تواند داشته باشد، یکی انجام یک فراخوانی بازگشتی دیگر(با آدرس بازگشت 'B') و دومی خود یک بازگشت است که با دستور goto START در خط 21 انجام می شود.

تابع هانوی بدون استفاده از پشته (به روش tail recursive) به شکل زیر خواهد شد:

START:

1. If(n==0)

2. return

3. Hanoi(n-1,f,h,t)

4. write(f->t)

5. n--

6. swap(h,t)

7. goto START

مثال بالا همان مثال هانوی با استفاده از Tail Recursion می باشد ولی در مثال زیر به جهت اینکه آخرین دستور حاصل جمع دو تابع بازگشتی است نمی توان آن را Tail Recursion نامید.

int comb(int n, int m)

1. if(m==n or m==0)

return 1 2.

3. return comb(m, n-1) + comb(m-1, n-1)

مثال: تابع بازگشتی comb زیر را به یک تابع غیر بازگشتی تبدیل کنید.

```
void RecComb(int m, int n, int &res)
{
1       int res1, res2;
2       if(m==n or m==0)
3       {
4               res = 1;
5               return;
6       }
7       RecComb(m, n-1, res1);
8       RecComb(m-1, n-1, res2);
9       res = res1 +res2;
}
```

حل:

```
int NonRecComb(int m, int n)
{
1       STACK s;
2       int res1, res2, res;
3       Char addr;
4       START:
```

```
5           if(m==n or m==0)
6           {
7                   res = 1;
8                   goto RET;
9           }
10          s.push(m, n, res, res1, res2, ‘A’);
11          n--;                                    //Parameter passing
12          goto START;
13  RET:
14          if(s.isEmpty())
15                  return res;
16          int temp = res;
17          s.pop(m, n, re1, res2, res, addr);
18          if(addr == ‘A’)
19          {
20                  res1 = temp;
21                  s.push(m, n, res, res1, res2, ‘A’);
22                  n--;
23                  m--;
24                  goto START;
25          }else{
```

```
26                  res2 = temp;
27                  res = res1 + res2;
28                  goto RET;
29          }
}
```