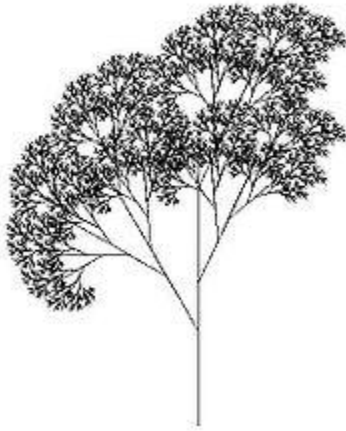
Рекурзија



Рекурзија е дефинирање на некој поим преку самиот себе, односно во дефиницијата се вклучува и поимот кој се дефинира.

Употреба на рекурзијата: дефиниција на мтематички функции, дефиниција на структури на податоци.

латински: **re** = назад + **currere** = извршува;

Да се случи повторно, со нови интервали. Многу математички функции може да се дефинираат рекурзивно:

- факториел
- фибоначиеви броеви
- Евклидов НЗД (најголем заеднички делител)
- аритметички операции

Концепт

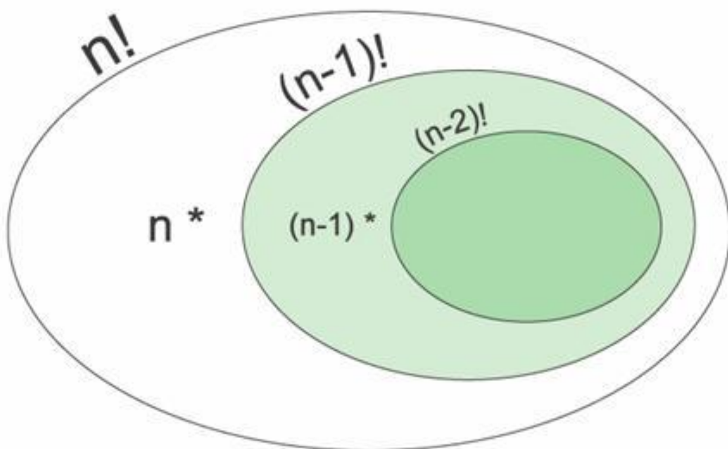
Рекурзијата е важен концепт во компјутерската наука бидејќи многу алгоритми можат со неа најдобро да се прикажат.

Кога се обидуваме да решиме некој проблем со помош на компјутерска програма, се трудиме да го разделиме проблемот на помали делови кои што се полесни за решавање. Често пишуваме посебни методи за да се справиме со овие потпроблеми. Кога потпроблемот е ист како и главниот, но на помало множество податоци, велиме дека проблемот е дефиниран рекурзивно.

Пример1. Фотографијата што ја држи момчево во рака е самата таа фотографија, но со помала големина.

Пример2. Факториел на даден број n.

n!=n*(n-1)!= n*(n-1) (n-2)!=...



Рекурзивни подалгоритми

Покрај повикувањето еден со друг, подалгоритмите можат да се повикуваат и самите себе. Таквото повикување се нарекува повратно (анг. recurrent) повикување, односно повторување (анг. recursion) на повикувањето. Затоа постапката е наречена рекурзија, а подалгоритмите се наречени рекурзивни подалгоритми.

Рекурзивните подалгоритми се многу елегантни и пократки од итеративните, но тешко разбирливи и побавни од нив. Тие можат да се реализираат како процедурални или како функциски подалгоритми.

Факториел

Најрепрезентативен пример на рекурзивен подалгоритам е пресметување на факториел на даден број n:

$$n! = \begin{cases} 1 \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots n = 0 \\ n * (n-1)! \ldots\ldots\ldots n > 0 \end{cases}$$

```
подалгоритам fact(n) ;
почеток
      ако n=0
            тогаш
                    fact <- 1
            инаку
                    fact <- n*fact(n-1);
крај
```

Времето на извршување на алгоритмот е $T(n) = \begin{cases} t1 \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots n = 0 \\ T(n-1) + t2 \ldots\ldots\ldots n > 0 \end{cases}$,

каде што t1 и t2 се константи.

За да се реши оваа рекурентна равенка применуваме техника на **последователна замена**.

Значи имаме:

T(n)=T(n-1)+t2

=(T(n-2)+t2)+t2

=T(n-2)+2t2

=(T(n-3)+t2)+2t2

=T(n-3)+3t2

…

Очигледно дека **T(n)=T(n-k)+kt2**, каде што 1<=k<=n.

Точноста на оваа релација може секогаш да се провери со индукција.

Ако n е познат, тогаш го повторуваме процесот на замена се додека не стигнеме до T(0) на десната страна од равенката. Но n не го знаеме, така што за да се добие T(0) на десната страна ставаме n-k=0, т.е. n=k.

T(n)=T(n-k)+kt2

=T(0)+nt2

=t1+nt2

Значи сложеноста на овој рекурзивен алгоритам за пресм

етување факториел е O(n).

Природен начин за решавање на факториел е пишување на рекурзивна функција која одговара на дефиницијата

```
Example 1.) n!

fact(non-negative integer n)
{
	if (n==0)
	{
		return 1;
	}
	else
	{
		return fact(n-1)*n;
	}
}
```

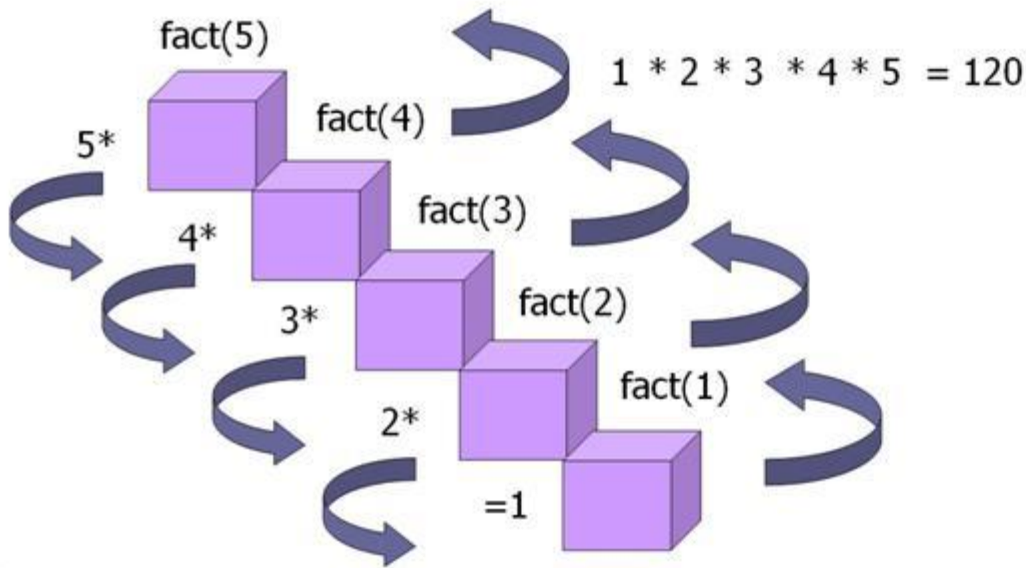
Оваа функција се повикува самата себе за да го пресмета следниот член. На крај ќе дојди до условот за прекин и ќе излези. Меѓутоа, пред да дојде до условот за прекин, таа ќе стави n повици на стек.

Постоењето на услов за прекин е неопходно кога се работи за рекурзивни функции. Ако се изостави, тогаш функцијата ќе продолжува да се повикува самата себеси се додека програмата не го наполни целиот стек, што ќе доведе до несоодветни резултати.

Како работи рекурзијата

При секое самоповикување на подалгоритмот натамошното извршување привремено се прекинува, освен при последното повикување кога целосно се извршува (тоа е граничниот случај). Потоа се продолжува секое прекинато извршување и тоа наназад.

За да се овозможи завршување на подалгоритмот по секое прекинато извршување, потребно е да се паметат вредностите на сите променливи во моментот на прекинувањето, и тоа за сите прекини. Во рекурзивните потпрограми тоа се постигнува со користење на посебен дел од меморијата т.н. стек. Во него се ставаат вредностите на променливите при секој прекин., а при продолжување на извршувањето се вадат.



За да се избегне бесконечното извршување на подалгоритмот:

- Мора да постои одреден критериум (краен критериум, **граничен случај**) за кој подалгоритмот не се повикува самиот себе.
- Секогаш кога подалгоритмот ќе се повика самиот себе мора да биде поблиску до крајниот критериум (граничен случај).

Рекурзијата во компјутерската наука е начин на размислување и решавање на проблеми. Всушност рекурзијата е една од централните идеи во компјутерската наука. Решавањето на проблем со помош на рекурзија значи дека решението зависи од решавањето на помалите инстанци на истиот проблем.

„Моќта на рекурзијата е во можноста на дефинирање на бесконечно множество на објекти со конечна состојба. Бесконечен број на сметања можат да се опишат со конечна рекурзивна програма што не содржи експлицитни повторувања".

Најмногу од комјутерско програмирачките јазици поддржуваат рекурзија дозволувајќи им на функциите да се повикуваат самите себе.

Општ метод на поедноставување е делење на проблемот на подпроблеми од ист тип. Ова е познато како „dialekting". Како и во компјутерската програмирачка техника, ова е познато како раздели па владеј и е клуч на дизјнот на многу важни алгоритми и е фундаментален дел на динамичкото програмирање.

Виртуелно, сите јазици за програмирање што се во употреба денес дозволуваат директна спецификација на рекурзивни функции и процедури. Секоја рекурзивна функција може да биде трансформирана во итеративна функција со користење на стек.

При креирањето на рекурзивна процедура битно побарување е дефинирањето на „основен случај" и потоа дефинирање на правила за решавање на многу покомплексни случаи со основниот случај. Клуч за рекурзивна процедура е тоа што при секој рекурзивен повик, доменот на проблемот мора да се намалува, се додека не се дојде до основниот случај.

Дефиниција: Алгоритамска техника каде функција, во обид да исполни одредена задача, се повикува себеси како дел од самата задача.

Забелешка: Секое рекурзивно решение е составено од два главни делови или случаи, притоа вториот дел има три компоненти.

- **Базен случај** – во кој проблемот е доволно едноставен да биде решен директно, и
- **Рекурзивен случај**. Рекурзивниот случај има три компоненти:
    - **Раздели** го проблемот на еден или повеќе поедноставни или помали делови на проблемот
    - **Повикај** ја функцијата (рекурзивно) за секој дел, и
    - **Комбинирај** ги решенијата на деловите во решение на целиот проблем.

Зависно од проблемот, секое од овие може да биде тривијално или комплексно.

Споредба на рекурзивни со итеративни подалгоритми

Од претходните примери се гледа дека при секое самоповикување на подалгоритмот, извршувањето се прекинува, освен при последното повикување кога целосно се извршува. Потоа, се продолжува секое прекинато извршување и тоа station наназад; прво последното прекинато извршување, потоа претпоследното итн. се до првото. За да се овозможи завршување на подалгоритмот по секое прекинато извршување, потребно е да се памети состојбата на сите променливи во моментот на прекинувањето, и тоа за сите прекини. Во рекурзивните потпрограми тоа се постигнува со користење на посебен дел од меморијата т.н. *стек*. Во него се ставаат вредностите на променливите при секој прекин, а при продолжување на извршувањето се вадат.

Рекурзивната постапка го скратува изворниот код на програмата, а во исто време го прави тешко разбирлив. Некои проблеми со рекурзивна постапка се решаваат доста лесно и елегантно, меѓутоа не треба да се сфати дека рекурзијата е начин за ефикасно програмирање.

Секое рекурзивно решение има и своја нерекурзивна варијанта.

Пример:

Проблем: Собери 1 000 000 денари за хуманитарни цели

Претпоставка: Секој е подготвен да донира по 100 денари

- ***Итеративно решение***

Посети 10 000 луѓе, барајќи им на сите по 100 денари

- ***Рекурзивно решение***

Ако е побарано од вас да дадете 100 денари, дајте му ги на човекот што ви ги побарал

Инаку

Посети 10 луѓе и побарај од секој од нив 1/10 од сумата што вам ви ја бараат да ја соберете

Собери ги парите што ви ги дале тие 10 луѓе и дај му ги на човекот што ви ги побарал

Евклидов алгоритам за наоѓање најголем заеднички делител (НЗД) на два цели броја

Се задава рекурзивно на следниот начин:

НЗД на два броја x и y е еднаков на НЗД од y и остатокот при делење на x со y т.е. НЗД(x, y)=НЗД(y, x mod y) (општ случај).

ако y=0 тогаш НЗД ја добива вредноста на x (граничен случај).

$$NZD(x, y)= \begin{cases} NZD(y, x \bmod y) \ldots\ldots\ldots\ldots y \neq 0 \\ x \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots y = 0 \end{cases}$$

Во Паскал:

```
function evklid(m:integer, n:integer):integer;

begin

  if n=0
  then
      evklid:=m;
      evklid:=evklid(n, m mod n);

end;
```

Повикувајќи ја оваа функција со evklid(314159, 271828) ги имаме овие рекурзивни (вгнездени) повици последователно:

```
evklid(271828, 42331)

evklid(42331, 17842)

evklid(17842, 6647)

evklid(6647, 4548)

evklid(4548, 2099)

evklid(2099, 350)

evklid(350, 349)

evklid(349, 1)

evklid(1, 0)


Значи НЗД е: 1
```

Кај евклидовиот алгоритам можевме да постапиме и вака, со што итеративно би го решиле проблемот:

```
function evklid(m:integer, n:integer):integer;
var pom:integer;
begin
  while n<>0 do
  begin
    pom:=n;
    n:=m mod n;
    m:=pom;
  end;
  evklid:=m;
end;
```

Рекурзивното решение е природно и се наметнува за решавање на математичките функции. За некои математички проблеми рекурзивното решение е и единствено. Таков пример е Акермановата функција зададена со релацијата:

$$A(n,m)=\begin{cases} m+1 \ldots\ldots\ldots\ldots n=0 \\ A(n-1,1) \ldots\ldots\ldots\ldots n\neq 0, m=0 \\ A(n-1, A(n, m-1)) \ldots\ldots n>0, m>0 \end{cases}$$

Можно е да се напишат едноставни рекурзивни програми кои се многу неефикасни. Пример, пресметување на првите n членови на Фибоначиевата низа зададена рекурзивно:

F(0)=F(1)=1 (гранични случаи)

```
F(n)=F(n-1)+F(n-2), n>=2
(рекурзивно правило)
F(n)=
```

$$\begin{cases} F(n-1)+F(n-2) \ldots\ldots n\geq 2 \\ 1 \ldots\ldots\ldots\ldots n=0 \\ 1 \ldots\ldots\ldots\ldots n=1 \end{cases}$$

Тоа се следните броеви: 1,1,2,3,5,8,13,21,34….

1. <u>Добро решение</u> (не користиме рекурзија).

```
function fibonaci(n:integer):integer;

var f0,f1,i,pom:integer;

begin

      f0:=1;

      f1:=1;

      if n=0 then fibonaci:=f0;

      if n=1 then fibonaci:=f1;

      for i=2 to n do
            begin

                  pom:=f0;

                  f0:=f1;

                  f1:=f1+pom;

            end;

            fibonaci:=f1;

end;
```

Очигледно дека сложеноста на овој алгоритам е **линеарна** O(n).

2. Лошо решение (користиме рекурзија на лош начин)

```
function fibonaci(n:integer):integer;

begin

      if n=0 then fibonaci:=1;

      if n=1 then fibonaci:=1;

      fibonaci:=fibonaci(n-1)+fibonaci(n-2);

end;
```

Кодот изгледа многу компактен и читлив, ама е исклучително неефикасен.

Се покажува дека сложеноста на алгоритмот во погорната имплементација е **експоненцијална**, а тоа е се разбира неприфатливо. Итеративното решение имаше **O(1)** сложеност.

Неефикасноста на ова решение има и свое интуитивно објаснување кое се добива кога ги пратиме вгнездените функциски повици. На пример да пресметаме fibonaci(5):

fibonaci(5)=fibonaci(4)+fibonaci(3)
fibonaci(5)=fibonaci(3)+fibonaci(2)+fibonaci(3)

Се дуплира пресметувањето за fibonaci(3).

Стратегија за работа со рекурзија

Два основни методи за работа со рекурзија се **раздели па владеј** (divide and conquer) и **динамичко програмирање** (dynamic programming).

· **Раздели па владеј (**анг. divide and conquer**)** -го дели проблемот на потпроблемите кои ги решава. Оваа метода функционира добро кога се потпроблемите независни. Меѓутоа кога ќе се примени директно на проблем чии потпроблеми не се независни како на пример горниот пример со Фибоначиевите броеви добиваме неефикасен алгоритам во кој се повторуваат беспотребни пресметувања. (пребројте колку пати пресметавме fibonaci(0) горе.)

· **Динамичко програмирање (**анг. dynamic programming**)** Постојат два типа

o **Од доле нагоре (bottom-up dynamic programming)** –ги пресметуваме по ред вредностите до зададените. Во секој чекор користиме веќе пресметани вредности со кои пресметуваме нови. Се разбира ова може да го употребиме ако имаме начин како да ги чуваме пресметаните вредности.

o **Од горе надоле (**top-down dynamic programming**)** –ја пресметуваме бараната вредност како во раздели па владеј методите, меѓутоа секоја конечно пресметана вредност ја запамтуваме и секојпат кога пресметуваме нова вредност користиме веќе пресметани вредности за да избегнеме повторни пресметки како кај лошото рекурзивно решение на Фибоначиевиот проблем.

o Во двата случаи може да се чуваат веќе пресметаните вредности во некое (доволно) големо поле или поврзана листа, кои ги користи функцијата што имплементира динамичко програмирање.

Рекурзија во поврзани листи

Ќе наведеме неколку примери на рекурзивни подалгоритми за поврзани листи. Ова е природно затоа што поврзаните листи и самите можат да се дефинираат рекурзивно. Да земеме дека секој јазел има своја нумеричка вредност.:

```
type jazel = record

     vrednost : integer;

     sleden : ^jazel;

end;
```

Сите подалгоритми наведени подоле користат заглавие, наречено анкер што покажува на почетокот од листата.

1. Одредување должина на листа.

За да се избројат јазлите во листата, треба да провериме дали анкерот покажува кон јазел или кон ништо (NIL). Ако покажува кон ништо, тогаш должината на листата е нула. А ако покажува кон јазел, тогаш броиме 1 за првиот јазел и продолжуваме да ги броиме јазлите од остатокот на листата.

Како и да е, покажувачот кон вториот јазел во листата е покажувач кон почеток на листа која е пократка од листата која ја набљудуваме. Броењето на јазлите во пократката листа е ист проблем како и тој што требаше да го решиме, само на помалку податоци. Значи го решаваме овој потпроблем со рекурзивно повикување на истиот метод.

Можеме да ја претставиме должината на листата на следниов начин:

Граничен случај: Ако анкерот покажува кон ништо, тогаш должината е 0.

Општ случај: Должината е 1 плус должината на листата без првиот јазел.

Во Паскал кодот би изгледал вака:

```
function dolzina(anker:^jazel):integer;
begin
    if anker=NIL then
        dolzina:=0
    else
        dolzina:=1+dolzina(anker^.sleden);
end;
```

Со мала измена можеме да ја пресметаме сумата на вредностите на јазлите:

```
function suma(anker:^jazel):integer;
begin
    if anker=NIL then
        suma:=0
    else
        suma:=anker^.vrednost+suma(anker^.sleden);
end;
```

2. Печатење на вредностите на листата.

```
procedure pecati(anker:^jazel);

begin

       if anker=NIL

       then

            break

       else

       begin

            writeln(anker^.vrednost);

            pecati(anker^.sleden);

       end;

end;
```

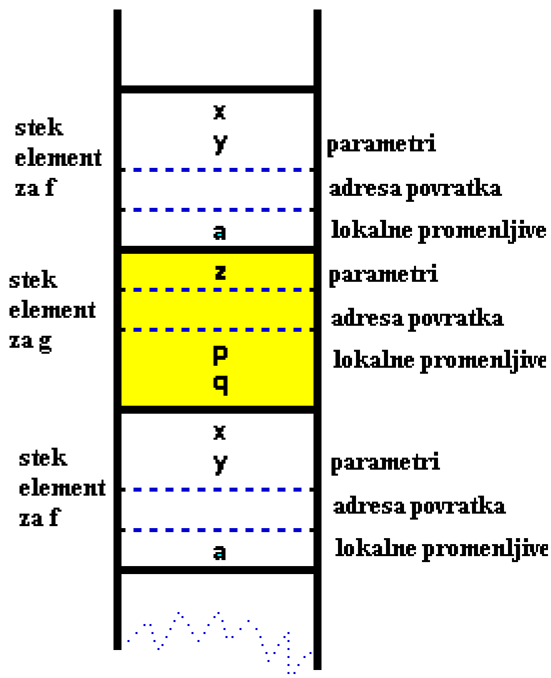
3. Печатење на вредностите на листата во обратен редослед.

Процедурата pecati се повикува рекурзивно дури не се посети последниот јазел. И потоа со враќање наназад од таму кај што биле прекинати се печатат вреднотите на листата во обратен редослед.

```
procedure pecati(anker:^jazel);

begin

     if anker=NIL

     then

          break

     else

     begin

          pecati(anker^.sleden);

          writeln(anker^.vrednost);

     end;

end;
```

По правило, модерните програмирачки јазици користат стек за повикување на работната меморија на сите повикани функции. Кога се повикува некоја процедура или функција, одреден број на повикувачки параметри се ставаат на стек. Кога се врши враќање од функцијата во повикувачкиот редослед, повикувачките параметри се вадат од стекот.

Кога функција повикува друга функција, најпрво се нејзините аргументи, адресата на враќање и конечно просторот за локалните променливи што се ставаат на стекот. Бидејќи секоја функција работи во сопственото опкружување или контекст, постои можност да функцијата се повика самата себе - значи како рекурзија. Оваа можност е екстремно корисна - бидејќи многу проблеми елегантно се решаваат на рекурзивен начин.



Стек, после извршување на неколку рекурзивни функции:

```
int f(int x, int y) {
int a;
    if ( услов_за_прекин ) return ...;
    a = .....;
    return g(a);
}

int g(int z) {
int p,q;
    p = ...; q = ...;
    return f(p,q);
}
```

Гледате дека функциите f и g односно нивните параметри и локални променливи се наоѓаат на стекот. Кога функцијата f се повика по втор пат од функцијата g, се креира нов повикувачки формат за вториот повик на функцијата f.

Структурите на податоци исто така можат рекурзивно да се дефинираат. Една од најважните класи на структури - дрвата, дозволуваат рекурзивни дефиниции кои водат до рекурзивни функции за нивна обработка.