Programiranje 2
2. predavanje

Saša Singer

singer@math.hr
web.math.pmf.unizg.hr/~singer

PMF – Matematički odsjek, Zagreb

Sadržaj predavanja

- Funkcije (kraj):
  - Rekurzivne funkcije (nastavak):
    - Particije.
    - Hanojski tornjevi.
- Struktura programa (prvi dio):
  - Blokovska struktura jezika.
  - Doseg varijable — lokalne i globalne varijable.
  - Vijek trajanja varijable, memorijske klase.
  - Program smješten u više datoteka. Vanjski simboli.

Informacije — odrada 5. predavanja

U petak, 3. 4., nema predavanja u redovitom terminu.

- Održava se Dan i noć na PMF–u, pa nema nastave.

Odrada 5. predavanja je sljedeći tjedan — za 3 dana, u:

- ponedjeljak, 16. 3., od 16–18 sati u (003).

Rekurzivne funkcije

Sadržaj

- Funkcije (kraj):
  - Rekurzivne funkcije (nastavak):
    - Particije.
    - Hanojski tornjevi.

Particije prirodnog broja

Particije — definicija

Particija (ili rastav) prirodnog broja $n \in \mathbb{N}$ je bilo koji rastav zadanog broja

- u zbroj pribrojnika koji su, također, prirodni brojevi,
- pri čemu poredak pribrojnika nije bitan.

Dakle, particija od n ima oblik

$$n = a_1 + a_2 + \cdots + a_m,$$

gdje je

- $m \in \mathbb{N} =$ broj pribrojnika u rastavu,
- $a_1, \ldots, a_m \in \mathbb{N}$ su pribrojnici.

Particije — zapis i broj

Obzirom na to da poredak pribrojnika nije bitan, tj.,

- dva rastava smatramo istim ako imaju iste pribrojnike, bez obzira na njihov poredak,

onda u zapisu možemo smatrati da su pribrojnici poredani — recimo, nepadajuće (ili nerastuće)

$$n = a_1 + a_2 + \cdots + a_m, \quad a_1 \leq a_2 \leq \cdots \leq a_m.$$

Zanimljivo je pronaći na koliko (različitih) načina se n može ovako zapisati (rastaviti).

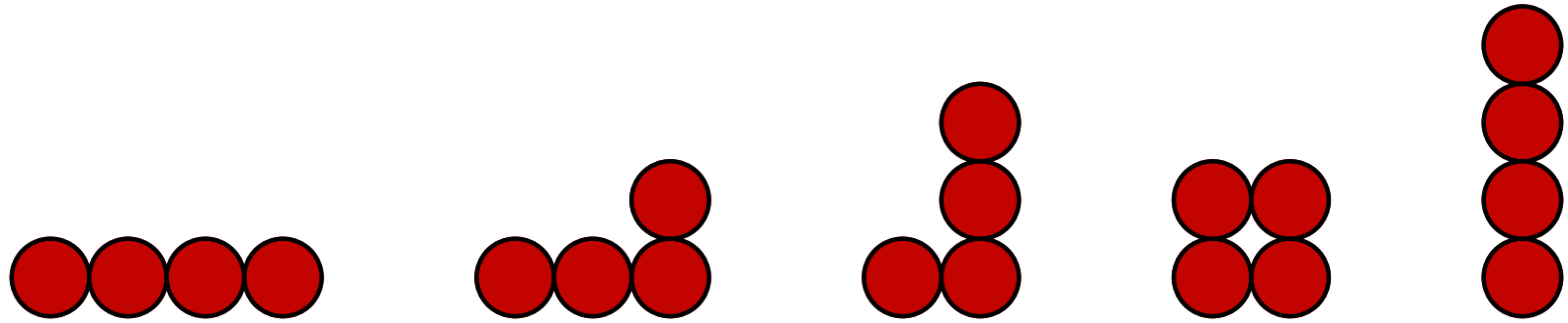
- Broj particija od n označavamo s $p(n)$.

Uočite da broj pribrojnika m može biti bilo koji (nije zadan).

Broj particija — zadatak

Primjer. Napišimo program koji učitava prirodni broj n i ispisuje broj particija $p(n)$ = broj rastava broja n u zbroj nepadajućih prirodnih brojeva.

Pokažimo koliko je takvih particija za $n = 4$.



$1+1+1+1$ $1+1+2$ $1+3$ $2+2$ 4

Dakle, broj particija je $p(4) = 5$.

Broj particija — ideja za rješenje

Trenutno, osim definicije, ne znamo ništa više o $p(n)$. Stoga je prirodna ideja za rješenje:

- generiraj, nekim redom, sve particije od n i izbroji ih.

Napomena. Ako tražimo samo broj particija $p(n)$, onda postoje i puno bolji načini za njegovo nalaženje (rekurzivne relacije i sl.).

Bez obzira na to, idemo realizirati gornju ideju, kao da je zadatak upravo

- generiranje svih particija od n.

Generiranje particija — razrada

Kako ćemo generirati sve particije od n?

- Pa, ..., nekim redom, jednu po jednu.

A kako bismo generirali jednu (neku) particiju od n?

$$n = a_1 + a_2 + \cdots + a_m, \quad a_1 \leq a_2 \leq \cdots \leq a_m.$$

To je već lakše:

- pribrojnik po pribrojnik, jedan za drugim,
- pazeći da pribrojnici ne padaju, tj. $a_\ell \geq a_{\ell-1}$, za $\ell \geq 2$.

Dakle, "izaberi" a_1, pa a_2, i tako redom.

Ako još pojedine pribrojnike a_ℓ "vrtimo" u petlji (po dozvoljenim vrijednostima), dobit ćemo i sve particije od n.

Generiranje particija — razrada (nastavak)

Ima samo jedan problem:

- broj pribrojnika m može varirati od 1 do n,

pa nema šanse da to napravimo "hrpom petlji" — jedna u drugoj ("broj petlji" varira).

Taj problem možemo riješiti rekurzivnom funkcijom. Ideja:

- Svaki poziv funkcije "vrti" samo jednu petlju — za svaki pojedini pribrojnik (onaj a_ℓ).

Rekurzivni pozivi realiziraju "petlje u petlji".

Naravno, nije baš očito kako bismo to trebali napraviti. Zato,

- krenimo od početka — prvog pribrojnika a_1,
- a onda — korak po korak (kao u indukciji).

Generiranje particija — prvi pribrojnik

Kako izabrati i "vrtiti" prvi pribrojnik a_1?

Zapišimo prvi korak u obliku:

$$n = a_1 + \text{“preostala suma”},$$

gdje "preostala suma" predstavlja zbroj preostalih pribrojnika $a_2 + \cdots + a_m$, ako ih ima.

- Pribrojnik a_1 možemo "vrtiti" u petlji od 1 do n — sve ove vrijednosti su dozvoljene.

Za svaku fiksnu vrijednost $a_1 =$ `i` znamo da mora biti

$$\text{“preostala suma”} = n - \texttt{i},$$

zato da ukupna suma bude zadani n.

Generiranje particija — skica koraka

I što sad, kad smo izabrali $a_1 =$ `i`?

- Preostaje nam problem da "preostala suma" $= n -$ `i` opet rastavimo u zbroj pribrojnika $(a_2 + \cdots + a_m)$.

Dakle, rekurzija se nazire.

- Zadana je tražena `suma` koju želimo dobiti — to će biti ulazni argument funkcije, zovimo ju `generiraj`.
- Na početku, u prvom koraku znamo (tražimo) da je `suma` $= n$. To će biti stvarni argument u vanjskom pozivu funkcije.
- Funkcija `generiraj(suma)` "vrti" sljedeći pribrojnik, zovimo ga `i`, u petlji od 1 do `suma`,
  - i zove samu sebe u obliku `generiraj(suma - i)`.

Generiranje particija — greška u koraku

Ali, oprez! Ovo je pogrešno, jer ne osigurava da pribrojnici ne padaju. Na primjer,

- fali nam uvjet $a_1 = \texttt{i} \leq a_2$ u prvom koraku.

Gdje je greška u razmišljanju?

Vratimo se na prvi korak. Kad izabremo pribrojnik $a_1 = \texttt{i}$,

- preostaje nam problem da $\texttt{suma} - \texttt{i}$ opet rastavimo u zbroj pribrojnika $(a_2 + \cdots + a_m)$.

Ali, taj "opet" nije isti problem kao i polazni!

- Ovom "novom" problemu moramo reći (zadati) od koje vrijednosti smije krenuti sljedeći pribrojnik (to je a_2 u prvom koraku).

Nazovimo tu vrijednost `prvi`.

Generiranje particija — popravak koraka

Potpuno isto vrijedi i u općem koraku!

- Funkcija `generiraj` mora dobiti još jedan ulazni argument `prvi`. To je polazna vrijednost za sljedeći pribrojnik (a ne 1).

Ovo je tzv. parametrizacija rekurzije.

Popravljena rekurzija ima sljedeći oblik:

- funkcija `generiraj(suma, prvi)` "vrti" sljedeći pribrojnik `i`, u petlji od `prvi` do `suma`,
  - i zove samu sebe u obliku `generiraj(suma - i, i)`.

Pripadni problem je: "generiraj sve particije broja `suma`, u kojima prvi (najmanji) pribrojnik mora biti veći ili jednak broju `prvi`".

Generiranje particija — popravak prvog poziva

Sad treba polazni problem "uložiti" u ovaj rekurzivni, tj. treba popraviti vanjski (prvi) poziv = podesiti stvarne argumente.

No, to je lako.

- U prvom koraku tražimo da je `suma` $= n$, a prvi pribrojnik smije početi od 1.

To će biti stvarni argumenti u vanjskom pozivu funkcije.

Dakle, vanjski poziv je `generiraj(n, 1)`.

Generiranje particija — kraj rekurzije

Ne zaboravimo: svaku rekurziju moramo nekako zaustaviti (prekinuti).

Naša rekurzija `generiraj(suma, prvi)` radi sljedeće:

- "generira sve particije broja `suma`, u kojima prvi pribrojnik mora biti veći ili jednak broju `prvi`".

Stvarno, a kad smo gotovi?

Odgovor: onda kad više nemamo što rastaviti!

- Ako je `suma` ≥ 1, onda "ima posla".
- Međutim, ako je `suma` $= 0$, onda smo gotovi — rastav je završen.

Ako još uočimo da uvijek vrijedi `suma` ≥ 0, sve je "čisto".

Razlog: u petlji u rekurziji je `i` $\leq$ `suma`!

Generiranje particija — korektnost rekurzije

Preciznije, vrijedi i jača relacija:

$$1 \leq \texttt{prvi} \leq \texttt{i} \leq \texttt{suma}.$$

To izlazi direktno iz "kôda" funkcije:

- funkcija `generiraj(suma, prvi)` "vrti" sljedeći pribrojnik `i`, u petlji od `prvi` do `suma`,
  - i zove samu sebe u obliku `generiraj(suma - i, i)`.

Iz gornje relacije je očito

$$0 \leq \texttt{suma} - \texttt{i} < \texttt{suma}.$$

To znači da rekurzivni pozivi funkcije `generiraj` strogo smanjuju prvi argument `suma`, pa se rekurzija sigurno prekida kad `suma` padne na 0 (tzv. korektnost rekurzije).

Generiranje particija — brojanje

Vratimo se početnom problemu — brojanju particija.

No, to je sad lako. U `generiraj` treba samo dodati brojanje generiranih particija.

- Zamjena posla: generiraj $\mapsto$ prebroji.

Pripadni problem je:

- "prebroji sve particije broja `suma`, u kojima prvi (najmanji) pribrojnik mora biti veći ili jednak broju prvi".

Realizacija je funkcija `particije(suma, prvi)`

- koja vraća broj svih takvih particija!

Generiranje particija — brojanje (nastavak)

Što točno radi `particije(suma, prvi)`?

- Ima brojač takvih particija, zovimo ga `broj`, kojeg inicijaliziramo na 0,
- "vrti" sljedeći pribrojnik `i`, u petlji od `prvi` do `suma`,
  - i zbraja vrijednosti koje vrate rekurzivni pozivi `particije(suma - i, i)`.

I sad oprez! Što radimo kad prekidamo rekurziju, tj. kad ulazna `suma` padne na 0?

- Vratimo vrijednost 1, jer to znači da smo tog trena "izgenerirali" jednu od traženih particija.

Generiranje particija — brojanje (nastavak)

Matematički rečeno, imamo rekurzivne relacije

$$\texttt{particije(suma, prvi)} = \sum_{\texttt{i=prvi}}^{\texttt{suma}} \texttt{particije(suma - i, i)}.$$

s tim da je "na dnu"

$$\texttt{particije(0, prvi)} = 1, \quad \text{za bilo koji } \texttt{prvi},$$

a "na vrhu" (za vanjski poziv — ono što tražimo)

$$p(n) = \texttt{particije(n, 1)}.$$

Ha!

Broj particija — funkcija

```
#include <stdio.h>

int particije(int suma, int prvi)
{
    int i, broj = 0;

    if (suma == 0) return 1;
        /* else */
    for (i = prvi; i <= suma; ++i)
            /* Sljedeci pribrojnik je i,
               rekurzivni poziv za suma - i. */
        broj += particije(suma - i, i);
    return broj;
}
```

Broj particija — glavni program

```
int main(void)
{
    int n;

    printf(" Upisi prirodni broj n: ");
    scanf("%d", &n);

    printf("\n Broj particija p(%d) = %d\n",
           n, particije(n, 1) );

    return 0;
}
```

Cijeli program je u `parts_1.c`.

Broj particija — trag poziva

"Trag poziva" možemo dobiti tako da na vrh funkcije dodamo ispis ulaznih vrijednosti.

```
int particije(int suma, int prvi)
{
    int i, broj = 0;

    printf("  suma = %d,  prvi = %d\n", suma, prvi);
    if (suma == 0) return 1;
    for (i = prvi; i <= suma; ++i)
        broj += particije(suma - i, i);
    return broj;
}
```

Broj particija — trag poziva za $n = 4$

```
suma = 4,  prvi = 1      (vanjski, bez pribrojnika)
suma = 3,  prvi = 1        1
suma = 2,  prvi = 1        1 + 1
suma = 1,  prvi = 1        1 + 1 + 1
suma = 0,  prvi = 1   ->   1 + 1 + 1 + 1
suma = 0,  prvi = 2   ->   1 + 1 + 2
suma = 1,  prvi = 2        1 + 2
suma = 0,  prvi = 3   ->   1 + 3
suma = 2,  prvi = 2        2
suma = 0,  prvi = 2   ->   2 + 2
suma = 1,  prvi = 3        3
suma = 0,  prvi = 4   ->   4
Broj particija p(4) = 5
```

Particije — malo priče

Inače, broj particija $p(n)$ broja n može se dobiti i analitički.

U formalnom redu

$$\begin{aligned}
&(1 + x + x^2 + x^3 + x^4 + \cdots) \\
&\qquad \cdot (1 + x^2 + x^4 + x^6 + \cdots) \\
&\qquad \cdot (1 + x^3 + x^6 + x^9 + \cdots) \cdots \\
&\qquad \cdot (1 + x^k + x^{2k} + x^{3k} + \cdots) \cdots \\
&= 1 + p(1)x + p(2)x^2 + p(3)x^3 + \cdots + p(n)x^n + \cdots
\end{aligned}$$

očita se koeficijent uz x^n. Zato je dogovorno $p(0) = 1$.

Na primjer, za $n = 100$ je $p(100) = 190\,569\,292$.

Particije — ispis svih particija

Zadatak. Za zadani $n \in \mathbb{N}$ treba ispisati sve particije broja n.

Uputa za rješenje. Pribrojnike treba spremati u neko polje (idealno je globalno i dinamički alocirano). ■

Napomena. Postoje i puno brži algoritmi za računanje broja particija $p(n)$, zadanog prirodnog broja n.

Bazirani su na rekurzivnim relacijama za $p(n)$. Na primjer, Eulerova rekurzija ima oblik

$$p(n) = \sum_{m=1}^{\infty} (-1)^{m-1} \left(p(n - \frac{m(3m-1)}{2}) + p(n - \frac{m(3m+1)}{2}) \right),$$

uz dogovor da je $p(n) = 0$ za sve $n < 0$. Potražite po webu!

Particije — varijacije problema

Do sada smo tražili broj particija $p(n)$, uz osnovni uvjet da u rastavu

$$n = a_1 + a_2 + \cdots + a_m$$

pribrojnici ne moraju biti različiti. Tako smo došli do uvjeta

$$a_1 \leq a_2 \leq \cdots \leq a_m.$$

U rastavu

$$n = a_1 + a_2 + \cdots + a_m,$$

na pribrojnike možemo postavljati različite dodatne uvjete,

- s tim da, i dalje, poredak pribrojnika nije bitan.

Particije — varijacije problema

Na primjer:

- Svi pribrojnici su međusobno različiti. Onda ih možemo strogo rastuće poredati

$$a_1 < a_2 < \cdots < a_m.$$

- Razlika susjednih pribrojnika je najmanje `k`, tj. mora biti

$$a_\ell + k \leq a_{\ell+1}, \quad \text{za} \quad \ell = 1, \ldots, m-1.$$

  Za $k = 1$, ovo odgovara prethodnom uvjetu $a_\ell < a_{\ell+1}$.

- U rastavu ima točno `k` pribrojnika, tj. $m = k$ je zadan.

Što treba promijeniti u našoj funkciji da dobijemo broj svih odgovarajućih particija?

Particije — izazov

Prethodno rješenje za funkciju `particije` nije baš efikasno, slično kao kod Fibonaccijevih brojeva.

- Za $n = 4$, $p(n) = 5$, a imamo 12 poziva funkcije.
- Za $n = 100$, $p(n) = 190\,569\,292$, a imamo $1\,642\,992\,568$ poziva funkcije.

Izazov. Promijenite funkciju tako da nema nepotrebnih rekurzivnih poziva, tj. tako da

- svaki rekurzivni poziv garantira da postoji barem jedna particija koju će taj poziv naći.

Uputa: Promjena je jednostavna — pogledajte gornju granicu granicu petlje po `i`, uz još malo opreza.

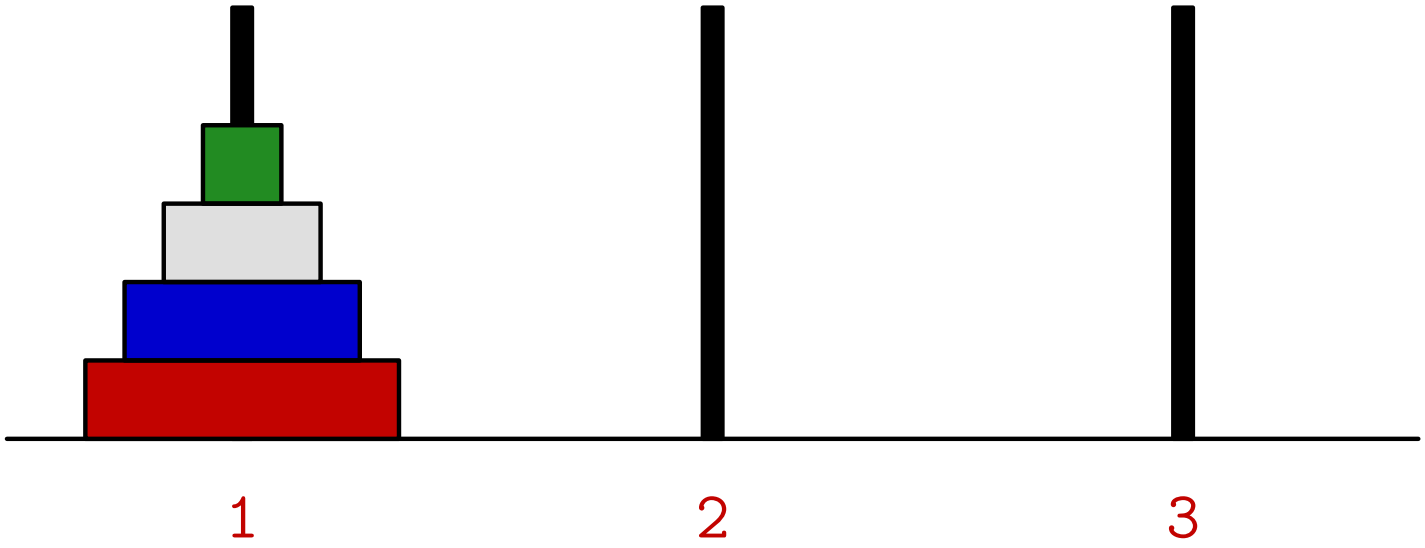
- Što treba staviti umjesto trenutne vrijednosti `suma`?

Hanojski tornjevi

Hanojski tornjevi

Primjer. Na štapu 1 nalazi se n diskova međusobno različitih veličina, poslaganih sortirano od najvećeg prema najmanjem (odozdo prema gore). Imamo još dva prazna štapa 2 i 3.

Na sljedećoj slici je primjer za `n = 4`.



Hanojski tornjevi — zadatak

Zadatak. Treba preseliti svih n diskova sa štapa 1 na štap 3, u minimalnom broju poteza, korištenjem pomoćnog štapa 2.

U prebacivanju diskova s jednog štapa na drugi treba poštovati sljedeća "pravila igre":

- u svakom potezu može se prebaciti samo jedan disk (najgornji na nekom štapu, i to na vrh nekog drugog),
- veći disk nikada se ne smije staviti iznad manjeg diska,
- manji (najgornji) disk se smije preseliti iznad bilo kojeg većeg diska na nekom drugom štapu, ili na prazan štap.

Dakle, prebacujemo "jedan po jedan" disk i smije samo manji na veći (ili na "ništa").

Hanojski tornjevi — razrada problema

Za početak, nije očito da problem uopće ima rješenje.

Uočiti da je prebacivanje jednog (najgornjeg) diska

- s nekog štapa (zovimo ga `odakle`),
- na neki drugi štap (zovimo ga `kamo`),

upravo = osnovni potez, kojeg uvijek znamo napraviti.

Ideja: svesti problem za n diskova

- na isti problem za malo manje diskova,
- koristeći osnovni potez za neki disk (ili neke diskove).

Dakle, želimo (rekurzivno) smanjivati n, dok ne dođemo do $n = 1$, a to opet znamo napraviti (osnovni potez).

Hanojski tornjevi — razrada (nastavak)

Traži se tzv. parametrizacija rekurzije — da zaista dobijemo više–manje isti problem s manjim brojem diskova.

Očito je da štapove treba uključiti u parametrizaciju, tj. da uloge štapova `odakle` i `kamo` moraju varirati.

Onda problem prebacivanja n diskova s `odakle` na `kamo` pišemo kao poziv funkcije

```
prebaci(n, odakle, kamo).
```

No, kad i n varira, zajedno sa štapovima, ključno pitanje je:

- kojih n diskova treba uzeti za "rekurzivna" prebacivanja,
- a na koji disk (ili diskove) treba primijeniti "osnovni" potez?

Hanojski tornjevi — razrada (nastavak)

Mogući izbor — izgleda dosta prirodno (na prvi pogled):

- makni najmanjeg s vrha štapa `odakle` na pomoćni štap, da "ne smeta",
- prebaci donjih $n-1$ s `odakle` na `kamo` ("rekurzija"),
- makni najmanjeg s pomoćnog na vrh od `kamo`.

Dakle, za "osnovni" potez uzmemo najgornji = najmanji disk.

Međutim, to jednostavno ne valja!

- Ovaj najmanji "blokira" pomoćni štap, jer ništa ne možemo staviti na njega,
- tj., nešto moramo vratiti na polazni štap.

A, zatim, postaje još kompliciranije ... i ne vidi se rekurzija.

Hanojski tornjevi — razrada (nastavak)

Pravi odgovor izlazi iz ograničenja "manji na veći".

- Rekurzivna funkcija `prebaci` prebacuje najgornjih n diskova na polaznom štapu, s `odakle` na `kamo`.

Uočite:

- Ostali diskovi ispod tih n (ako ih ima) su sigurno veći, pa ne mogu "smetati" kod prebacivanja najgornjih n!
- Do tih najgornjih n možemo doći bez dodatnih poteza.

U realizaciji funkcije `prebaci`, ključni korak je

- prebaciti najdonji = najveći disk s `odakle` na `kamo`, na njegovo pravo mjesto — to je traženi "osnovni" potez!

Naravno, prvo treba doći do najdonjeg, a zatim sve ostale vratiti na njega — "rekurzivna" prebacivanja $n - 1$ diskova.

Hanojski tornjevi — razrada (nastavak)

Problem Hanojskih tornjeva za gornjih n diskova svodi se na:

- prebaci gornjih $n-1$ diskova na pomoćni štap,
- prebaci jedan disk na odredišni štap — to je najdonji(!) disk od polaznih n,
- prebaci onih gornjih $n-1$ diskova s pomoćnog na odredišni štap.

Bitna stvar za korektnost algoritma:

- kod izvođenja rekurzivnih poziva za gornjih $n-1$ diskova,
- ti diskovi su sigurno manji od n-tog (najdonjeg),
- pa uvijek mogu na njega, tj. on ne može "zasmetati".

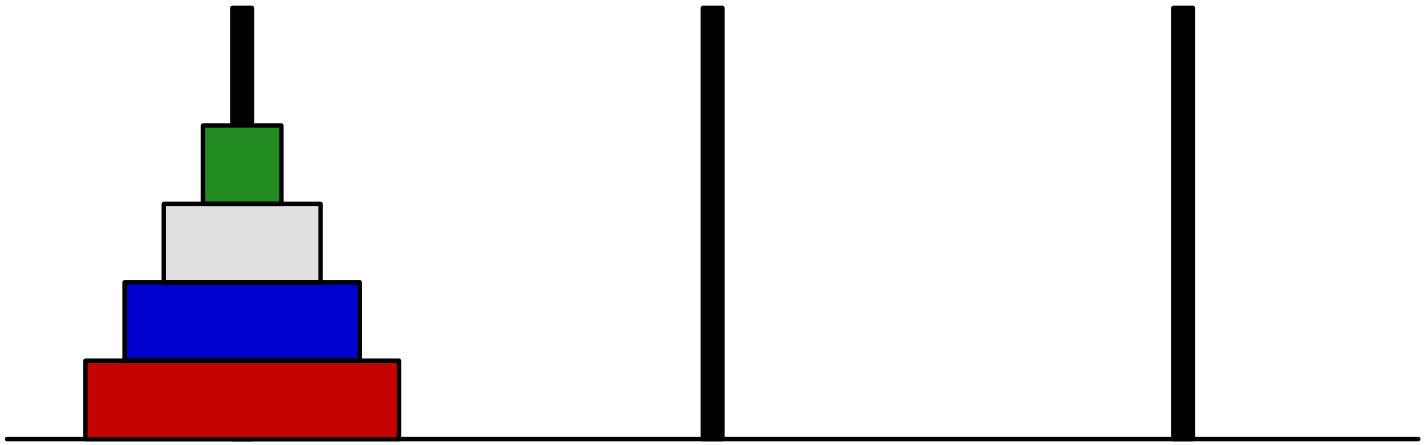
Dakle, sigurno ide "manji na veći"!

Hanojski tornjevi — razrada (nastavak)

Grafički, to izgleda ovako:

1. korak:

```
n = 4, prebaci gornja 3 na pomoćni
```

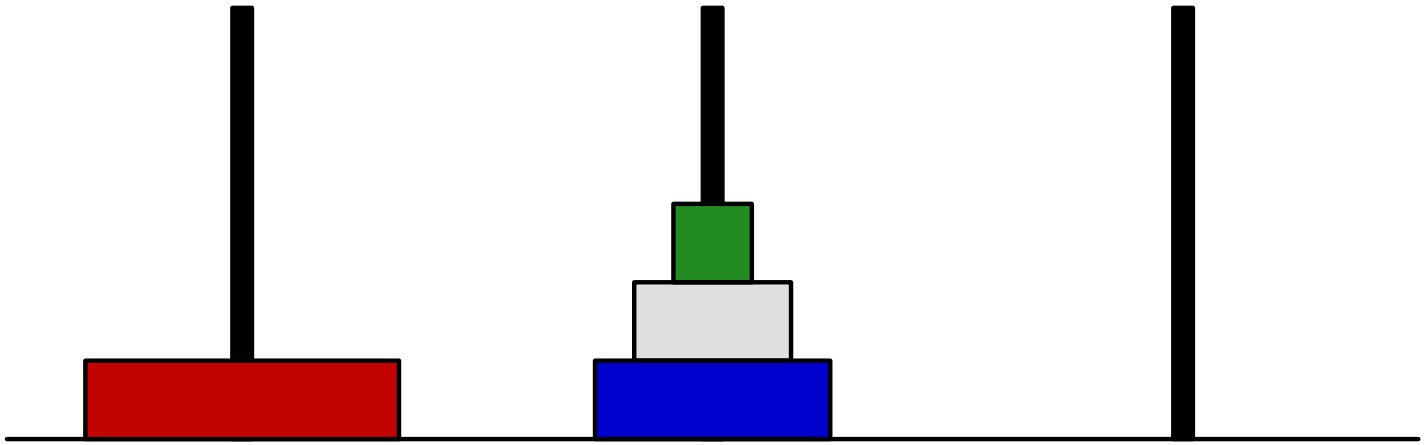


Hanojski tornjevi — razrada (nastavak)

Grafički, to izgleda ovako:

2. korak:

```
prebaci najveći na kamo
```

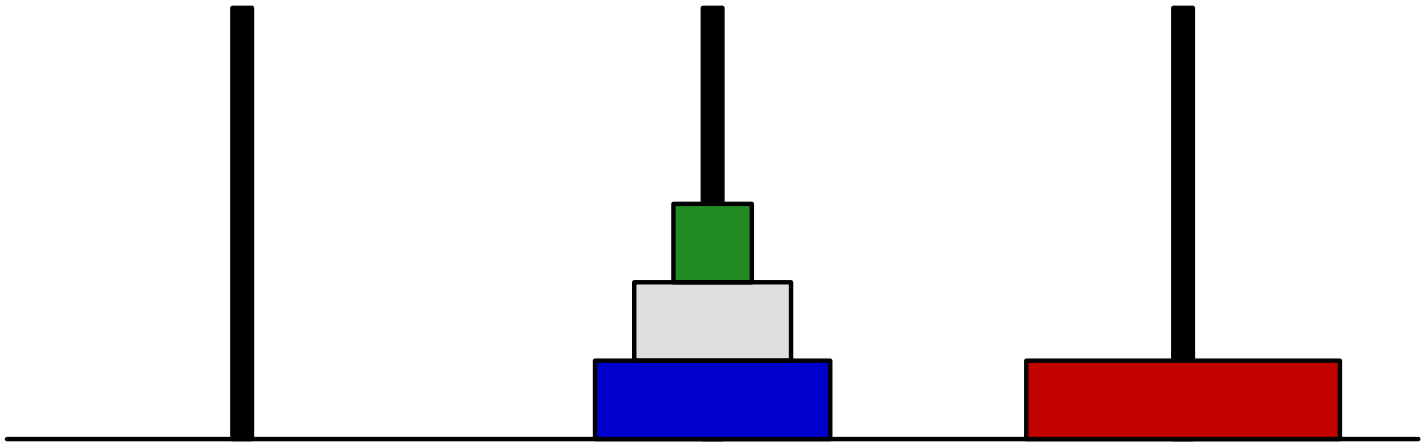


Hanojski tornjevi — razrada (nastavak)

Grafički, to izgleda ovako:

3. korak:

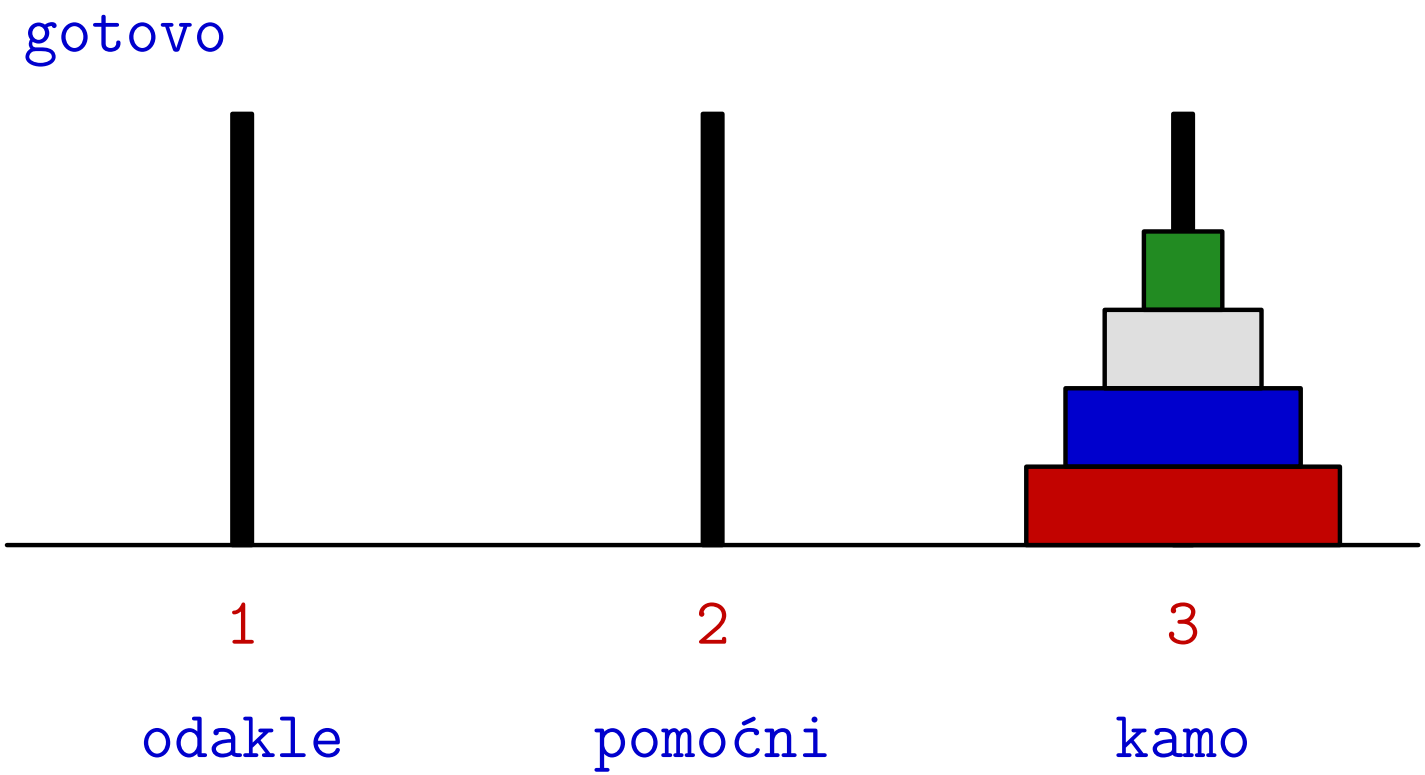
```
prebaci gornja 3 na kamo
```



Hanojski tornjevi — razrada (nastavak)

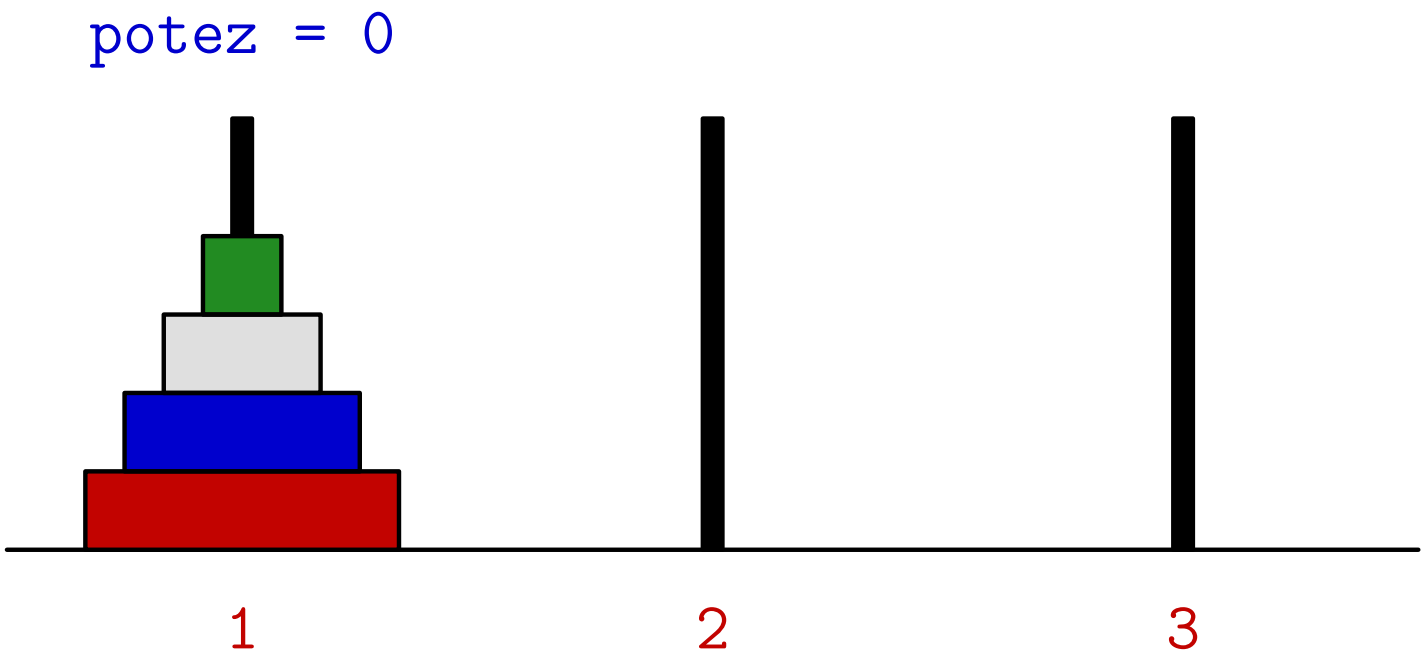
Grafički, to izgleda ovako:

Nakon 3. koraka:



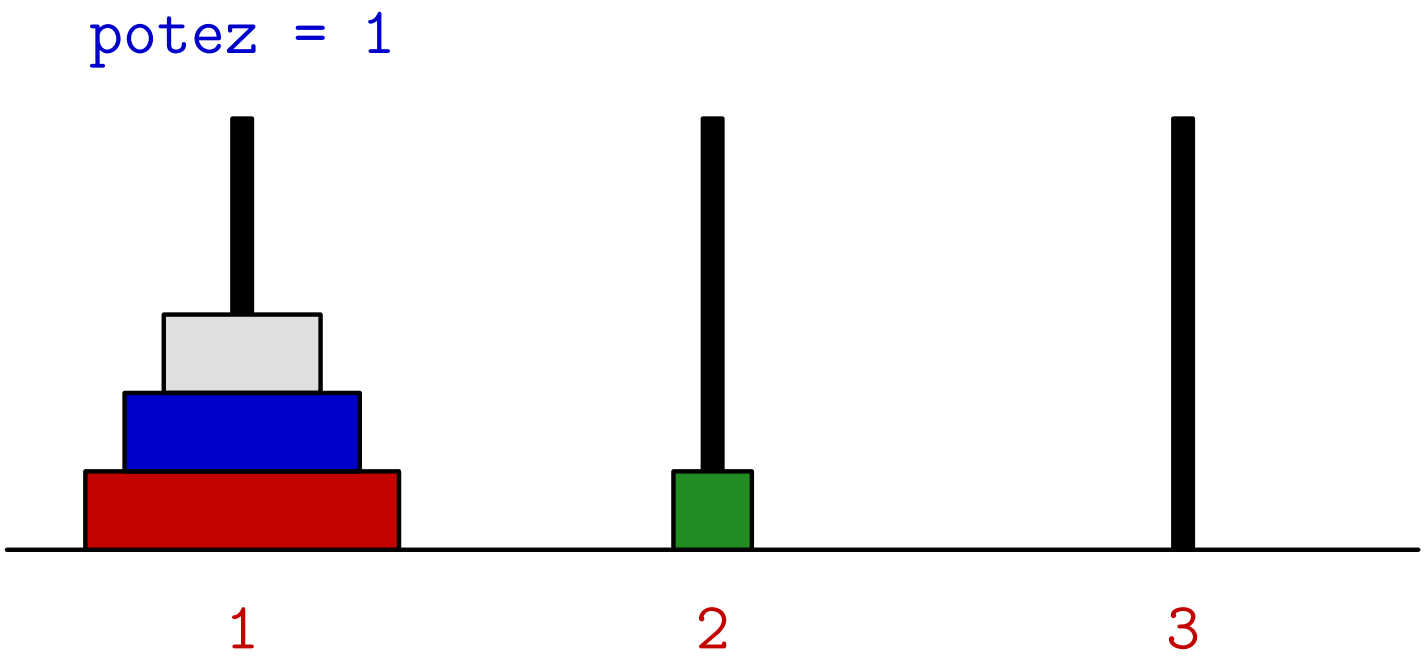
Hanojski tornjevi — potezi

Ovako izgleda prebacivanje potez po potez.



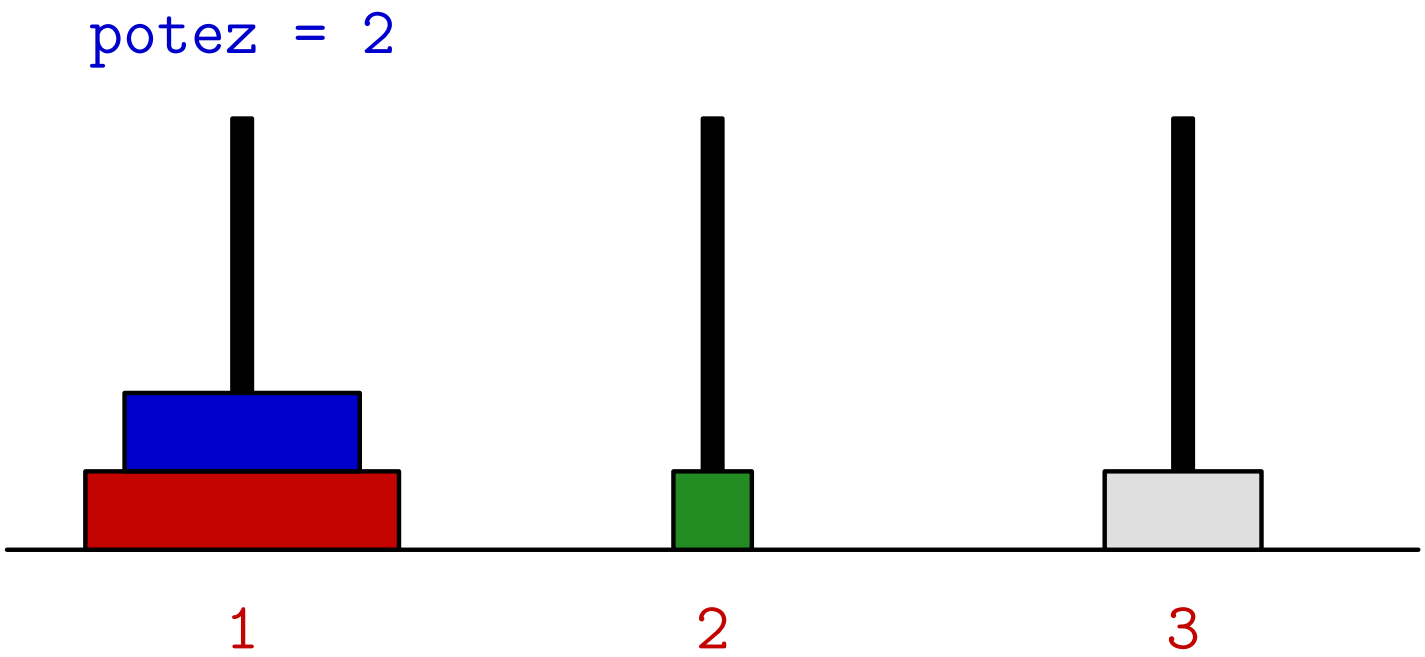
Hanojski tornjevi — potezi

Ovako izgleda prebacivanje potez po potez.



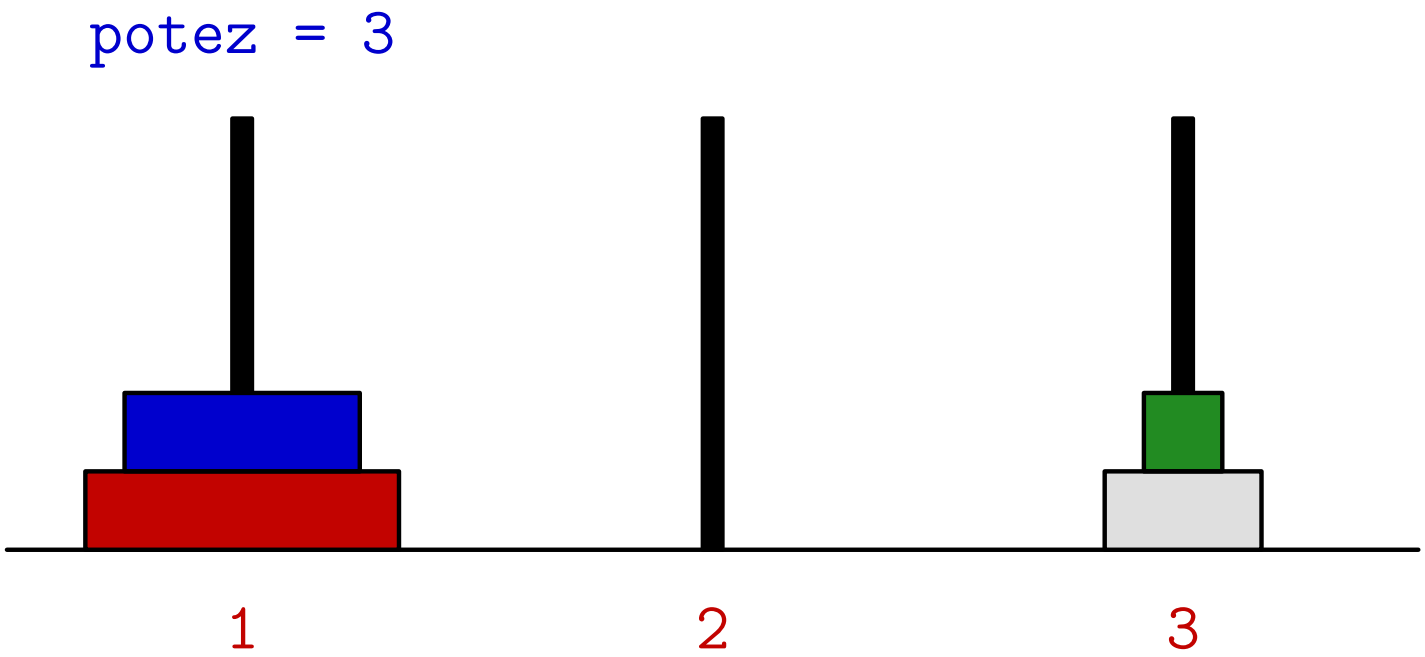
Hanojski tornjevi — potezi

Ovako izgleda prebacivanje potez po potez.



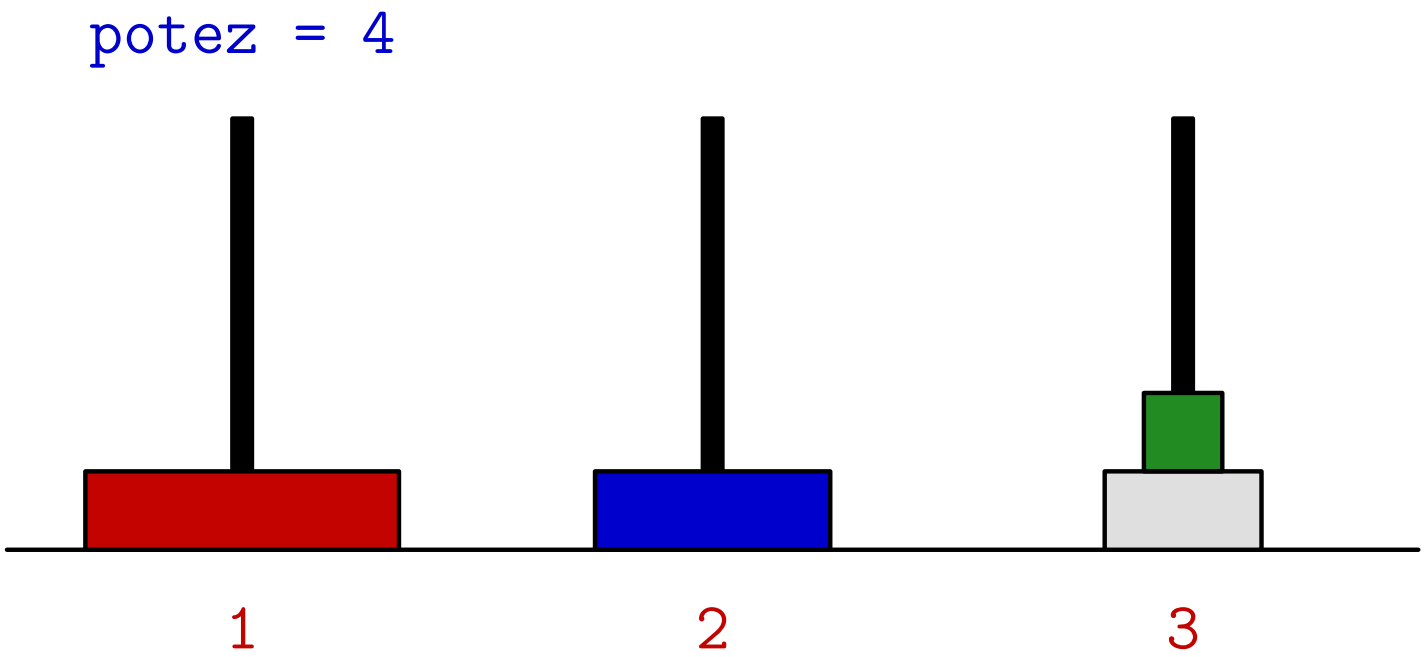
Hanojski tornjevi — potezi

Ovako izgleda prebacivanje potez po potez.



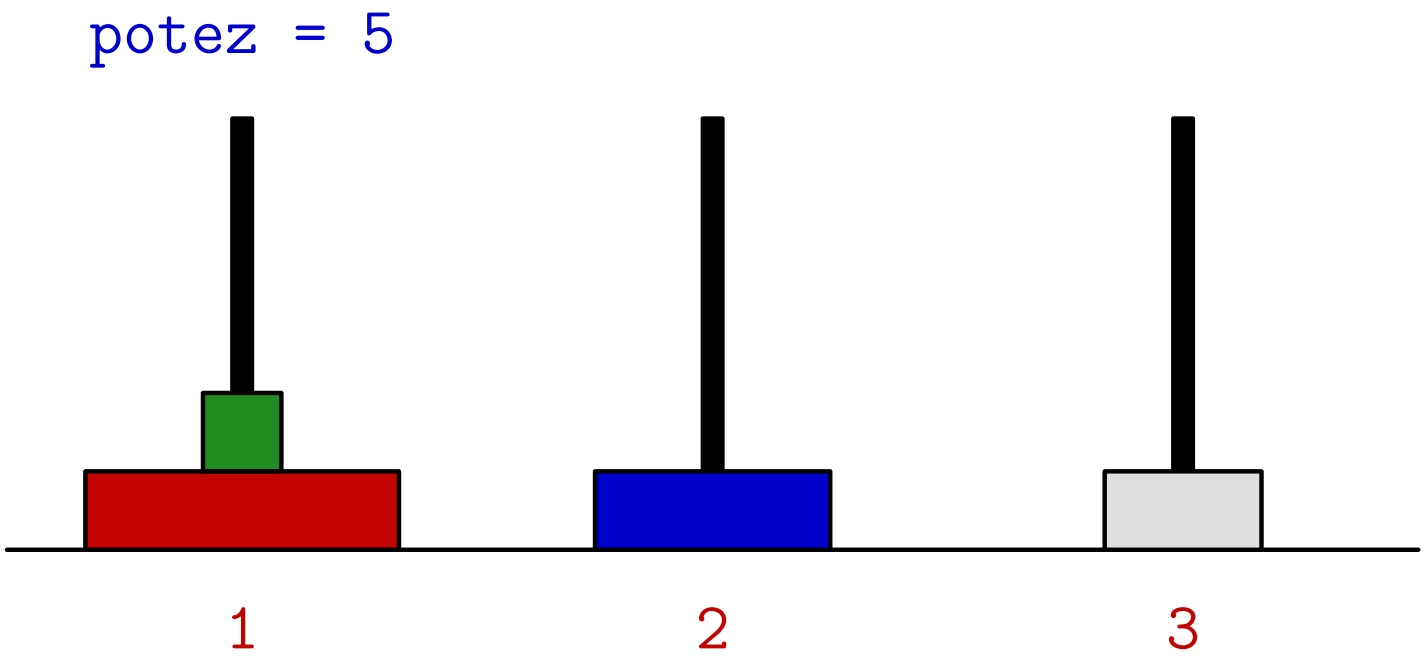
Hanojski tornjevi — potezi

Ovako izgleda prebacivanje potez po potez.



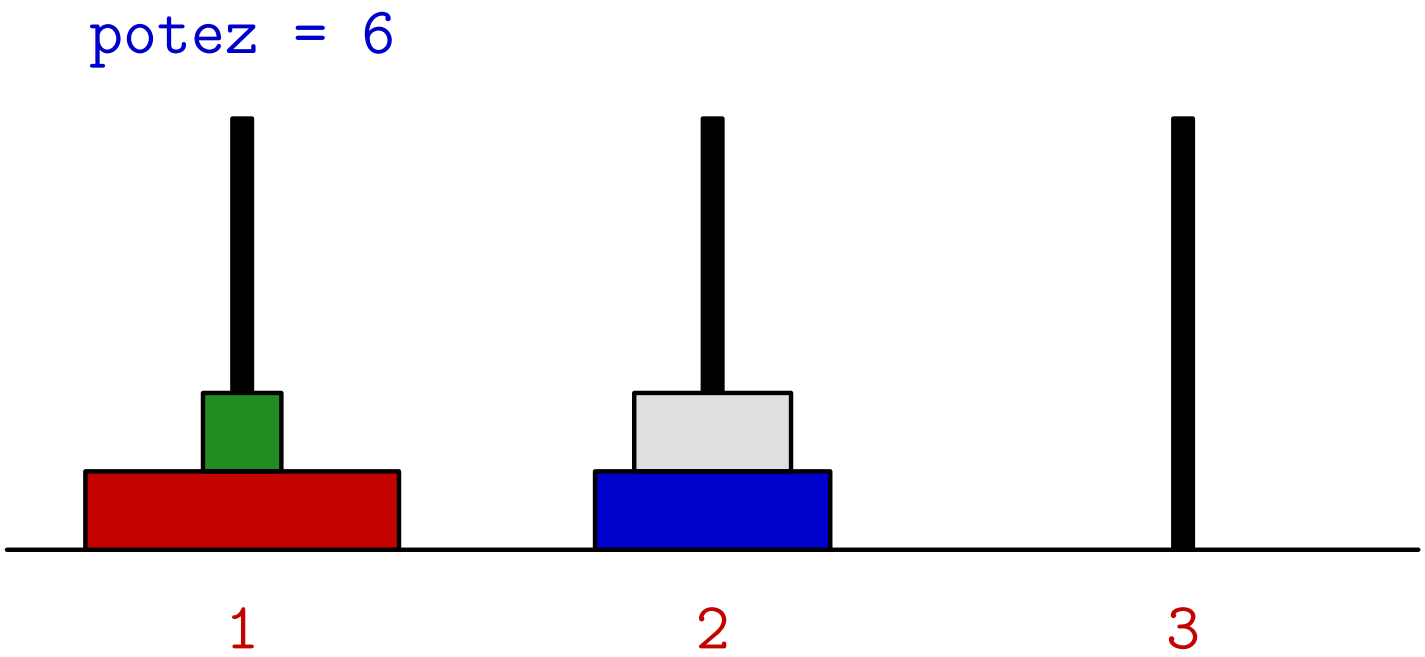
Hanojski tornjevi — potezi

Ovako izgleda prebacivanje potez po potez.



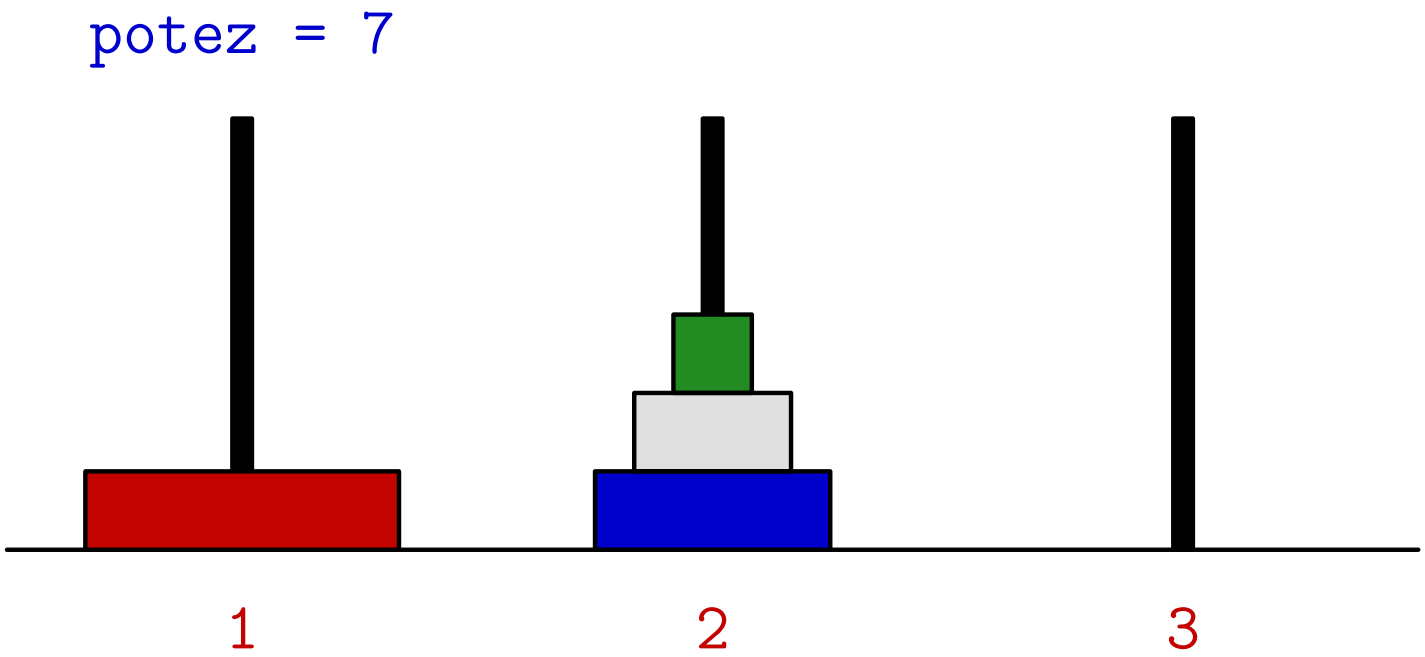
Hanojski tornjevi — potezi

Ovako izgleda prebacivanje potez po potez.



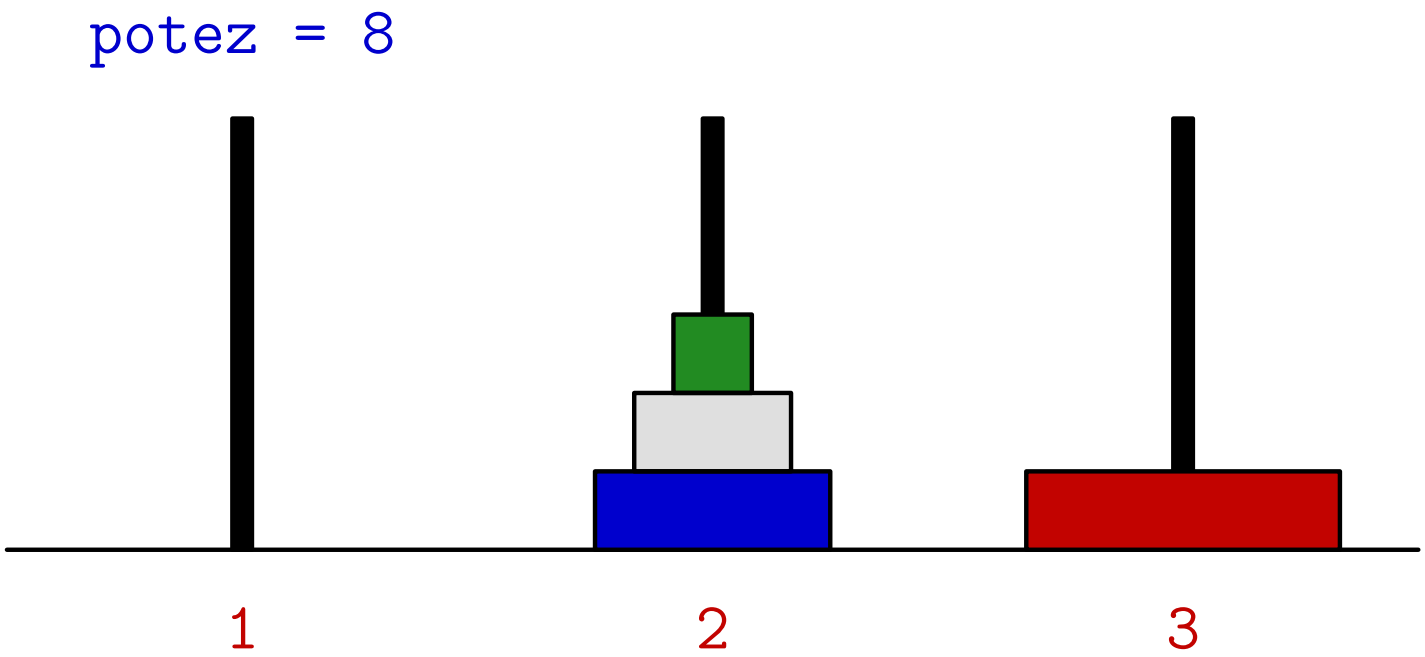
Hanojski tornjevi — potezi

Ovako izgleda prebacivanje potez po potez.



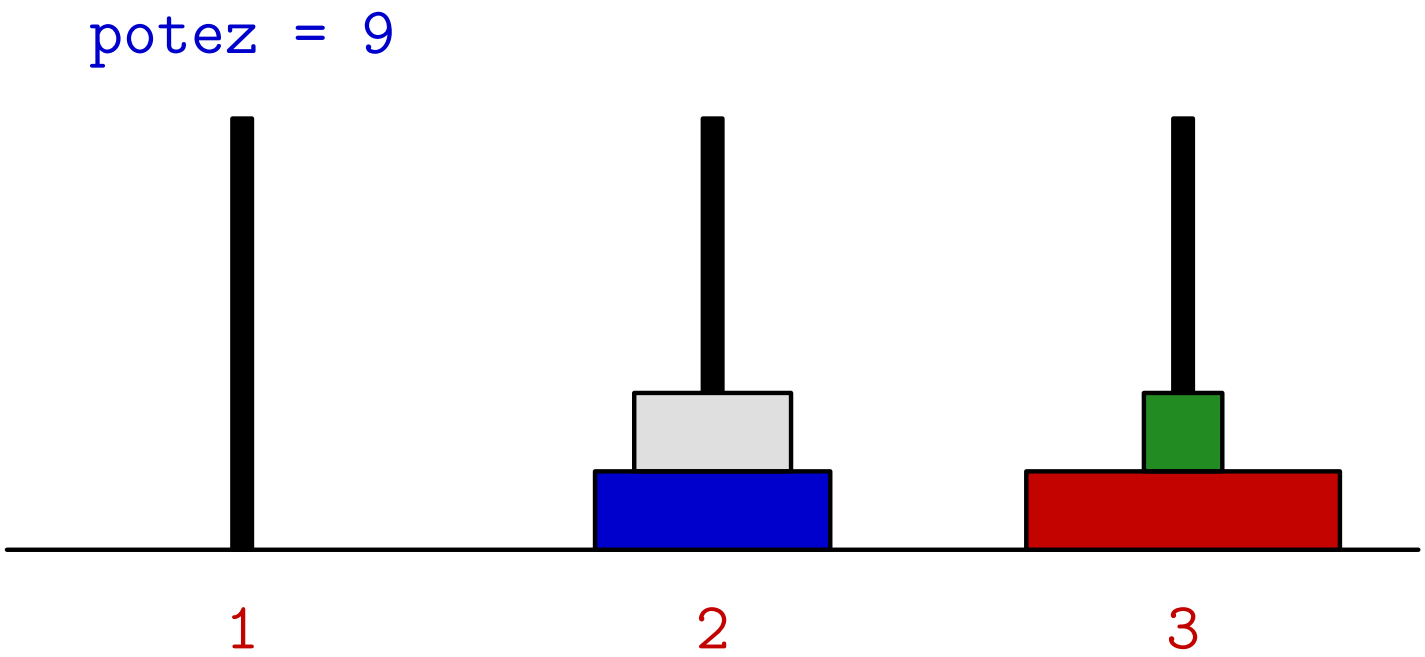
Hanojski tornjevi — potezi

Ovako izgleda prebacivanje potez po potez.



Hanojski tornjevi — potezi

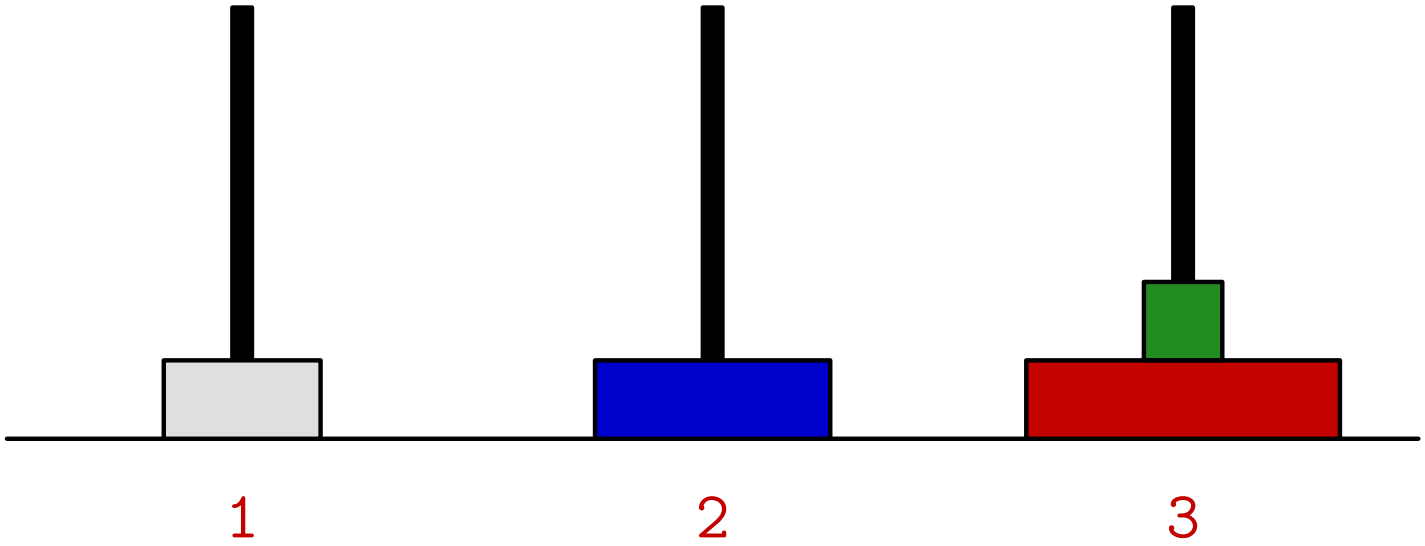
Ovako izgleda prebacivanje potez po potez.



Hanojski tornjevi — potezi

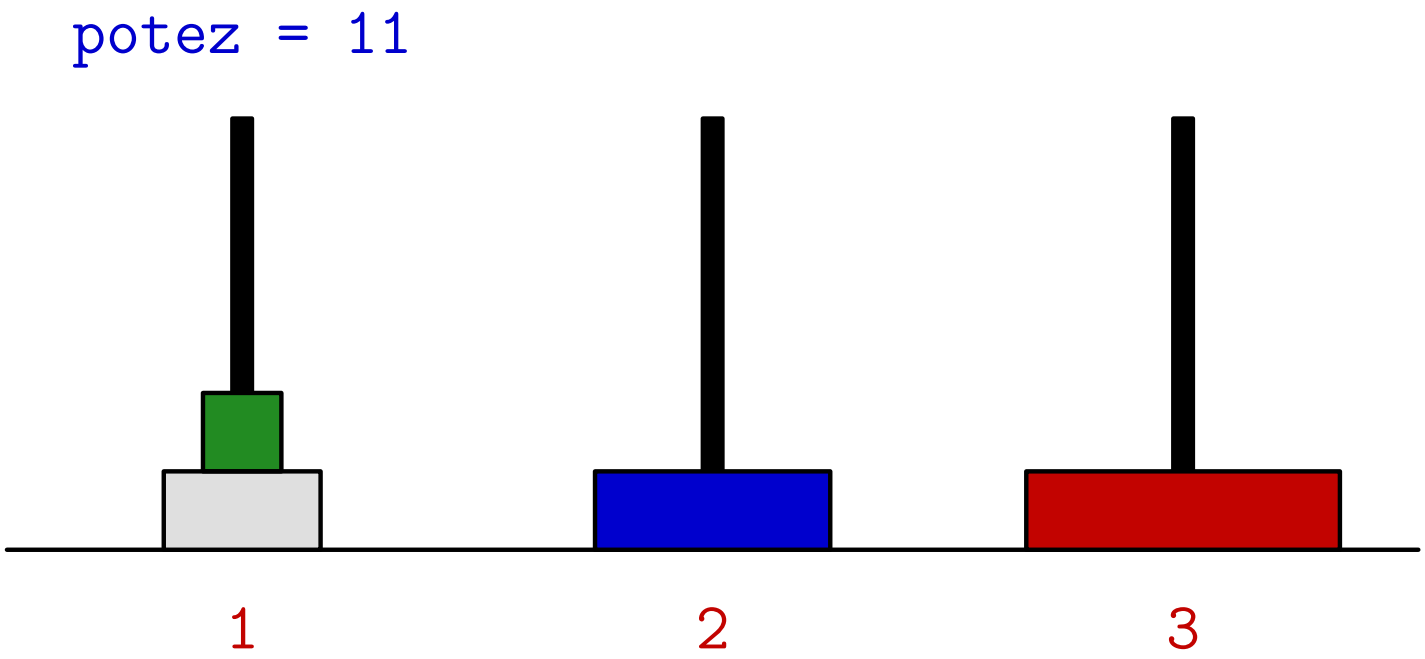
Ovako izgleda prebacivanje potez po potez.

```
potez = 10
```



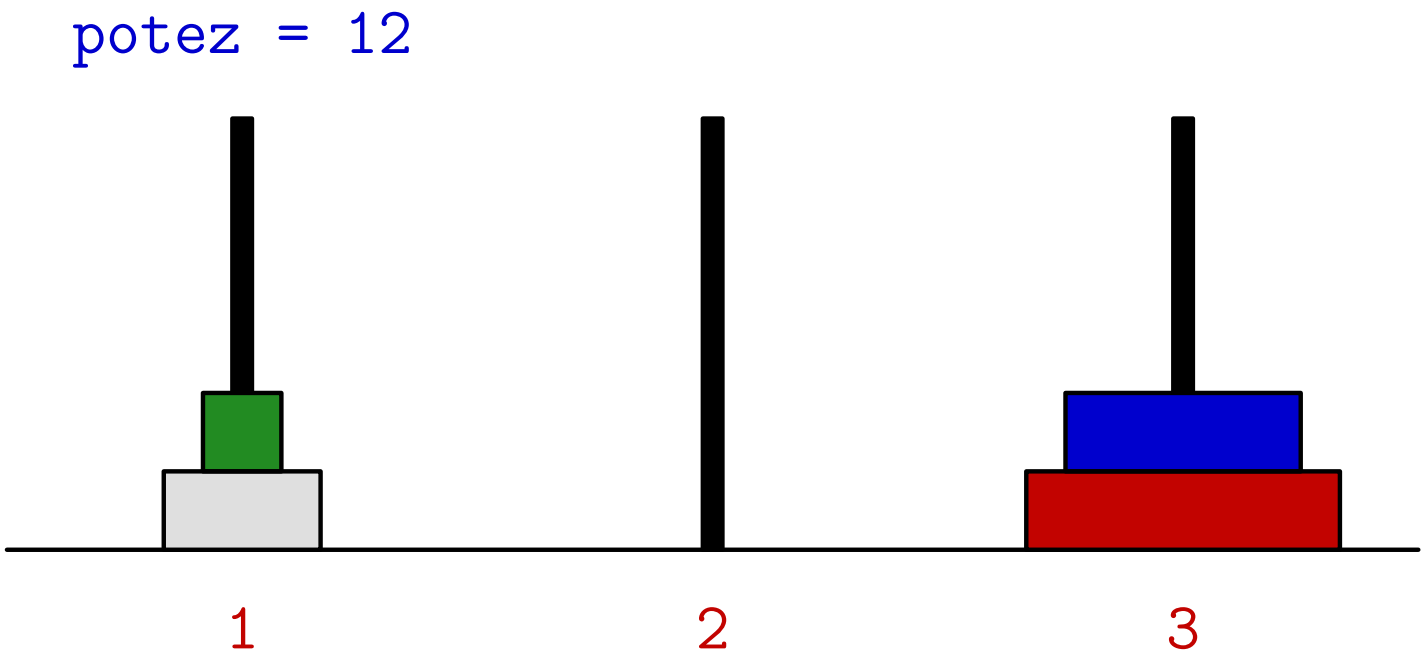
Hanojski tornjevi — potezi

Ovako izgleda prebacivanje potez po potez.



Hanojski tornjevi — potezi

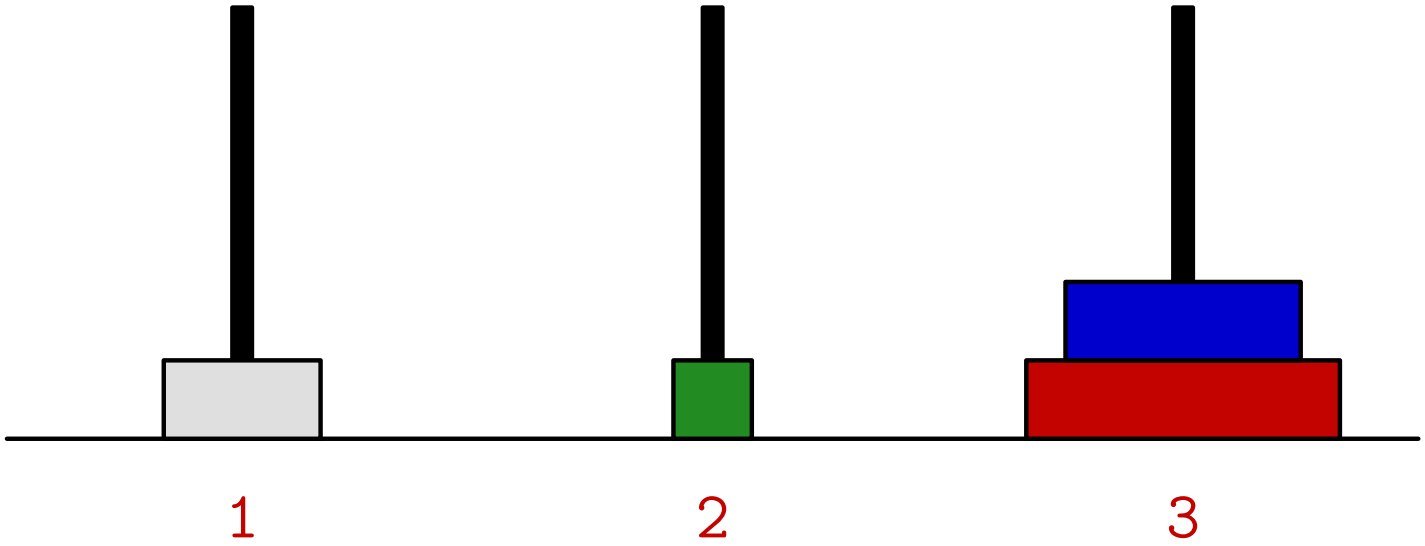
Ovako izgleda prebacivanje potez po potez.



Hanojski tornjevi — potezi

Ovako izgleda prebacivanje potez po potez.

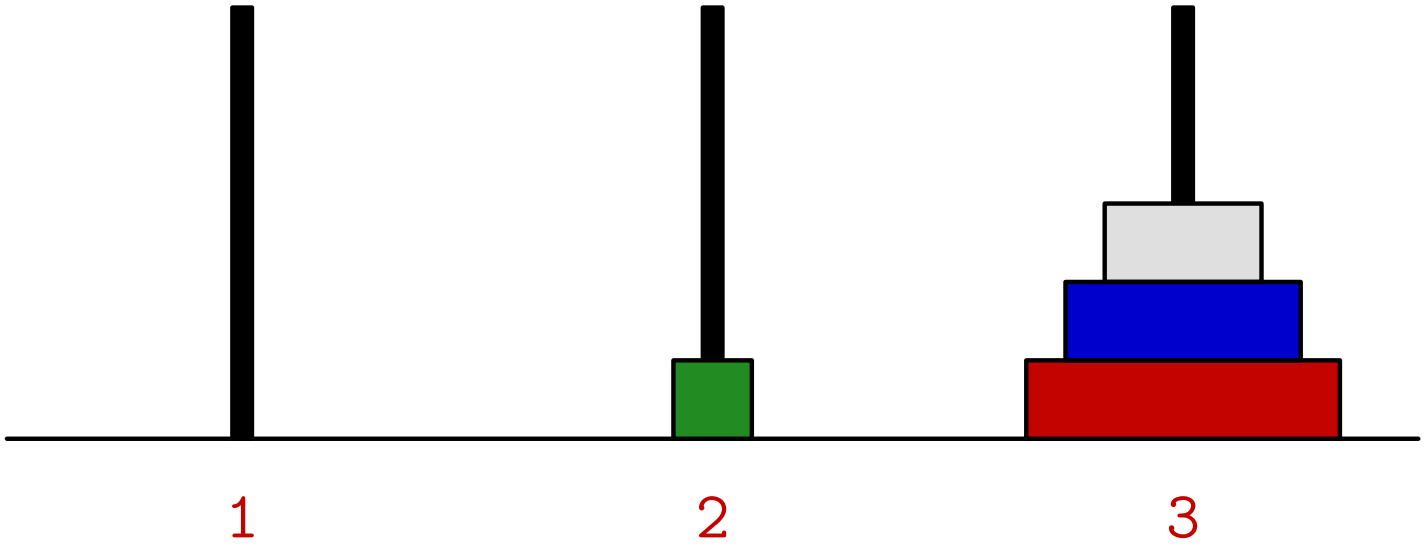
```
potez = 13
```



Hanojski tornjevi — potezi

Ovako izgleda prebacivanje potez po potez.

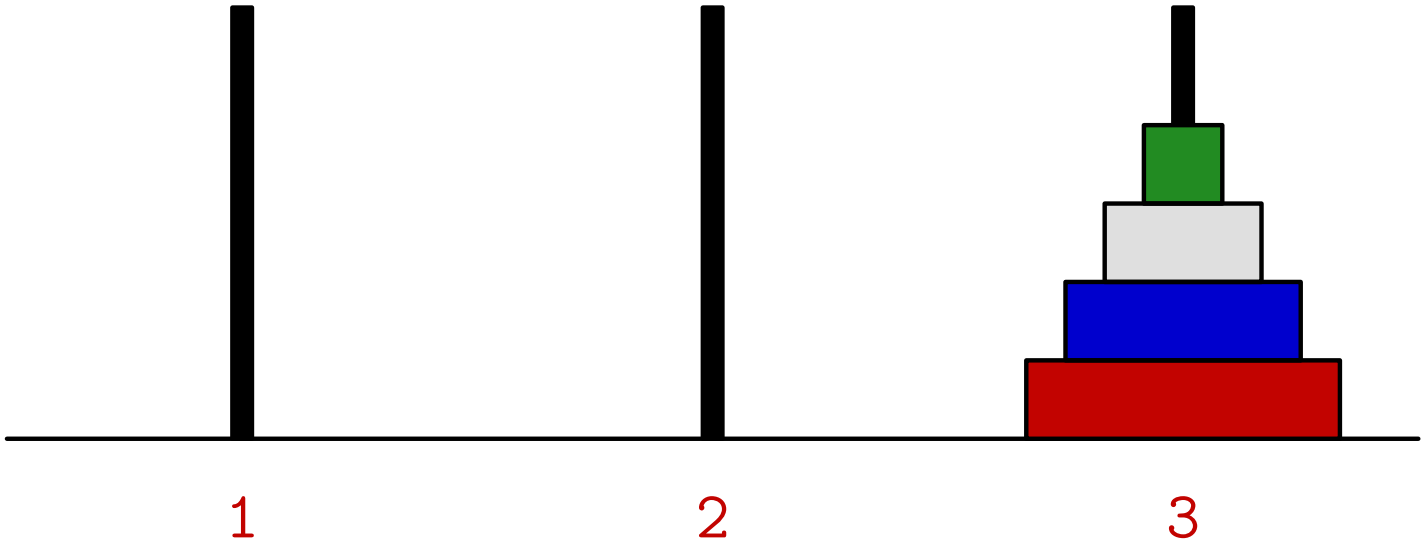
```
potez = 14
```



Hanojski tornjevi — potezi

Ovako izgleda prebacivanje potez po potez.

```
potez = 15
```



Hanojski tornjevi — animacija

Zgodnu animaciju Hanojskih tornjeva možete naći na web–stranici:

`https://www.mathsisfun.com/games/towerofhanoi.html`.

Hanojski tornjevi — skica programa

Program za rješenje problema Hanojskih tornjeva, očito, ima dvije funkcije:

- `prebaci_jednog` — koja realizira osnovni potez prebacivanja jednog (najgornjeg) diska s nekog na neki drugi štap,
- `Hanojski_tornjevi` — koja realizira rekurzivno prebacivanje gornjih n diskova (za zadani n) u obliku:
  - prebaci gornjih $n-1$ diskova na pomoćni štap,
  - prebaci jedan disk (najdonji od n) na odredišni štap,
  - prebaci onih gornjih $n-1$ diskova s pomoćnog na odredišni štap.

Precizna realizacija ovih funkcija ovisi o tome što želimo dobiti kao rješenje problema (tj. što je izlaz).

Hanojski tornjevi — redoslijed poteza

Imamo dvije osnovne mogućnosti:

- želimo točan redoslijed svih osnovnih poteza ("potez po potez"),
- želimo samo ukupni broj osnovnih poteza.

Realizirat ćemo prvu varijantu koja piše redoslijed poteza. Drugu — probajte sami.

Obje funkcije tada moraju dobiti točna značenja pojedinih štapova:

- `prebaci_jednog` mora znati `odakle` i `kamo` prebacuje,
- `Hanojski_tornjevi`, osim `n`, isto mora znati `odakle` i `kamo` prebacuje. Radi jednostavnosti, dodat ćemo i `pomocni`, da ga ne moramo računati iz preostala dva.

Hanojski tornjevi — početak programa

Sljedeći program ispisuje redosljed poteza, tj. prebacivanja (najgornjih) diskova. Program poziva rekurzivnu funkciju `Hanojski_tornjevi`.

Funkcija `prebaci_jednog` samo ispisuje osnovni potez.

```
#include <stdio.h>

void prebaci_jednog(int odakle, int kamo)
{
    printf("  prebaci s %d na %d\n", odakle, kamo);

    return;
}
```

Hanojski tornjevi — rekurzivna funkcija

```
void Hanojski_tornjevi(int n, int odakle,
                       int kamo, int pomocni)
{
  if (n > 0) {
    Hanojski_tornjevi(n - 1, odakle, pomocni, kamo);
    prebaci_jednog(odakle, kamo);
    Hanojski_tornjevi(n - 1, pomocni, kamo, odakle);
  }
  return;
}
```

Kraj rekurzije: za $n = 0$ ne radimo ništa — nema poteza!

Nažalost, i ti "prazni" pozivi troše vrijeme (ima ih 2^n, v. iza).

Hanojski tornjevi — efikasnija (brža) funkcija

```
void Hanojski_tornjevi(int n, int odakle,
                       int kamo, int pomocni)
{
  if (n <= 1)    // Uz n > 0, to znaci n == 1.
    prebaci_jednog(odakle, kamo);
  else {
    Hanojski_tornjevi(n - 1, odakle, pomocni, kamo);
    prebaci_jednog(odakle, kamo);
    Hanojski_tornjevi(n - 1, pomocni, kamo, odakle);
  }
  return;
}
```

Kraj rekurzije: za $n = 1$ radimo osnovni potez!

Hanojski tornjevi — glavni program

```
int main(void) {
  int n;

  for (n = 1; n <= 5; ++n) {
    printf("\n Prebaci %d diskova s 1 na 3:\n", n);
    Hanojski_tornjevi(n, 1, 3, 2);
  }

  return 0;
}
```

Broj diskova n "vrtimo" od 1 do 5 (`hanoi_a0.c`, `hanoi_a1.c`).

Zadatak. Dodajte ovom programu i brojač osnovnih poteza.

Hanojski tornjevi — broj poteza

Pretpostavimo da imamo n diskova i neka je h_n broj osnovnih poteza (prebacivanja po jednog diska).

Iz razrade algoritma odmah vidimo da je

$$h_n = h_{n-1} + 1 + h_{n-1} = 2h_{n-1} + 1, \quad \text{za} \quad n \geq 1,$$

i $h_0 = 0$ (pa je $h_1 = 1$, kao što i očekujemo).

Ova nehomogena diferencijska (ili rekurzivna) jednadžba može se riješiti (to se uči na višim godinama) i njezino rješenje je

$$h_n = 2^n - 1, \quad \text{za} \quad n \geq 0.$$

Zadatak. Dokažite da je ovo zaista rješenje gornje jednadžbe! Dokažite da je broj "praznih" poziva u prvoj funkciji 2^n.

Hanojski tornjevi — komentari

Primijetite da naš algoritam, usput, dokazuje da polazni problem za n diskova

- uvijek ima rješenje, za svaki n,
- i da je minimalni broj poteza jednak $2^n - 1$,

tj. optimalno rješenje je, zapravo, i jedinstveno.

Vidimo da broj poteza u premještanju diskova vrtoglavo raste.

- Razlog: Imamo samo 3 štapa, tj. samo jedan pomoćni.

Zanimljivo poopćenje ovog problema dobivamo ovako:

- Imamo k štapova, gdje je $k \geq 3$ unaprijed zadan, tako da imamo više pomoćnih štapova na raspolaganju.

Za $k > 3$, rješenja ima više, a traži se minimalni broj poteza.

Hanojski tornjevi — ograničenje

Zadatak. U "ograničenom" problemu Hanojskih tornjeva, i dalje imamo samo 3 štapa, uz dodatno ograničenje:

- u svakom potezu, disk se smije prebaciti s nekog štapa samo na susjedni štap,
- tj. zabranjeno je izravno prebacivanje s prvog na treći štap, ili obratno.

Ekvivalentno: u svakom potezu, ili stavljamo disk na srednji štap, ili mičemo disk sa srednjeg štapa.

Sastavite algoritam koji rješava ovaj problem (i tako dokažite da on ima rješenje). Koliki je broj poteza za n diskova?

Rješenje. Minimalni broj poteza $= 3^n - 1$.

Struktura programa

Sadržaj

- Struktura programa (prvi dio):
  - Blokovska struktura jezika.
  - Doseg varijable — lokalne i globalne varijable.
  - Vijek trajanja varijable, memorijske klase.
  - Program smješten u više datoteka. Vanjski simboli.

Struktura C programa

C program je, zapravo,

- skup definicija objekata — varijabli i funkcija:
  - varijable "modeliraju" ili zauzimaju memoriju, a
  - funkcije "modeliraju" ili sadrže instrukcije.

Komunikacija između funkcija ide preko:

- argumenata i vrijednosti koje vraćaju funkcije,
- vanjskih ili globalnih varijabli — to su one definirane izvan bilo koje funkcije.

Funkcije mogu biti

- u bilo kojem poretku u izvornom programu,

a program može biti smješten (rastavljen) u više datoteka (sve dok ne "cijepamo" funkcije, što je zabranjeno).

Struktura C programa (nastavak)

Objekti u C programu mogu biti:

- globalni ili vanjski (engl. external) — definirani izvan bilo koje funkcije, ili
- lokalni ili unutarnji (engl. internal) — što znači da su definirani "lokalno" unutar neke funkcije.

Bitno:

- Funkcije u C-u su uvijek globalne ili vanjske,

jer C

- ne dozvoljava da se funkcije definiraju unutar neke druge funkcije

(za razliku od nekih drugih jezika, poput Pascala).

Struktura C programa (nastavak)

Vanjski objekti imaju svojstvo da se

- svaka referenca na takav objekt s istim imenom zaista i odnosi na istu stvar, tj.
- isto ime ne može značiti dvije različite stvari!

Takva imena su, u načelu, tzv. vanjski ili "javni" simboli, a

- pripadni objekti su univerzalno dohvatljivi (dostupni),

čak i kad su funkcije, koje ih dohvaćaju, smještene u raznim datotekama (i prevode se odvojeno).

Ova univerzalna dohvatljivost može se ograničiti (ključnom riječi `static` za vanjske objekte, v. malo kasnije).

Struktura C programa (nastavak)

Za unutarnje objekte to ne vrijedi — oni nisu univerzalno dohvatljivi.

Funkcije ne mogu biti unutarnji objekti, tj.

- jedini unutarnji objekti su varijable,

ili preciznije,

- argumenti i varijable definirani unutar funkcija

(argumenti su, ionako, lokalne varijable).

Krenimo od lokalnih varijabli — njih je najlakše objasniti,

- jer se definiraju unutar blokova,

kao što smo dosad, uglavnom, i radili.

Blokovska struktura jezika — lokalne varijable

Blok naredbi je svaki niz naredbi koji se nalazi unutar vitičastih zagrada (recimo, tijelo funkcije).

- C dozvoljava da se u svakom bloku naredbi deklariraju varijable. Takve varijable zovu se lokalne varijable.
- Deklaracija varijabli unutar bloka mora prethoditi prvoj izvršnoj naredbi u bloku (standard C90, ne mora u C99).

Primjer:

```
if (n > 0) {
    int i;    /* deklaracija varijable */
    for (i = 0; i < n; ++i)
    ...
}
```

Blokovska struktura jezika — doseg varijable

Pravila dosega ("vidljivosti" ili dostupnosti):

- Varijabla definirana unutar nekog bloka vidljiva je samo unutar tog bloka (samo tamo postoji).
- Izvan bloka toj varijabli se ne može pristupiti (ona ne postoji), tj. njezino ime izvan bloka nije definirano.
- Izvan bloka može biti deklarirana varijabla istog imena, ali ona je nedostupna unutar bloka, jer je "prekrivena" varijablom istog imena.
- Varijabla definirana izvan bloka vidljiva je u tom bloku ako nije "pokrivena" lokalnom varijablom istog imena, tj. ako u bloku nije definirana varijabla istog imena.

Ukratko, dostupna je samo "najlokalnija" varijabla istog imena.

Blokovska struktura jezika — doseg varijable

Primjer:

```
int main(void) {
    int x, y;
    ...
    if (x > 0)
        {
        double y;  /* int y NIJE vidljiv
                      u bloku */
                   /* int x JE vidljiv u bloku */
        ...
        }
}
```

Funkcije — doseg formalnog argumenta

Formalni argument funkcije vidljiv je unutar funkcije i nije dohvatljiv (definiran) izvan nje.

- Doseg (vidljivost) formalnog argumenta je isti kao i doseg lokalne varijable definirane na početku funkcije.

Primjer:

```
int x, y;
...
void f(double x) {
    double y;  /* int x i int y NISU        */
    ...        /* vidljivi unutar funkcije */
}
int main(void) { ... }
```

Atributi varijable

Varijabla je ime (sinonim) za neku lokaciju ili neki blok lokacija u memoriji (preciznije, za sadržaj tog bloka).

Sve varijable imaju tri atributa: tip, doseg (engl. scope) i vijek trajanja (engl. lifetime).

- Tip = kako se interpretira sadržaj tog bloka (piše i čita, u bitovima), što uključuje i veličinu bloka.
- Vijek trajanja = u kojem dijelu memorije programa se rezervira taj blok.
  - Stvarno, postoje tri bloka: statički, programski stog ("run–time stack") i programska hrpa ("heap").
- Doseg = u kojem dijelu programa je taj dio memorije "dohvatljiv" ili "vidljiv", u smislu da se može koristiti — čitati i mijenjati.

Atributi varijable (nastavak)

- Prema tipu, imamo varijable tipa
  - `int`, `float`, `char`, itd.
- Prema dosegu, varijable se dijele na:
  - lokalne (unutarnje) i globalne (vanjske).
- Prema vijeku trajanja, varijable mogu biti:
  - automatske i statičke (postoje još i dinamičke).

Doseg i vijek trajanja određeni su, u principu, mjestom deklaracije, odnosno, definicije objekta (varijable) — unutar ili izvan neke funkcije.

"Upravljanje" vijekom trajanja (a, ponekad, i dosegom) vrši se tzv. identifikatorima memorijske klase (ključne riječi `auto`, `extern`, `register` i `static`) i to kod deklaracije objekta.