Programiranje 2

Predavanje 02 - rekurzivne funkcije

Matej Mihelčić

Prirodoslovno-matematički fakultet
Matematički odsjek

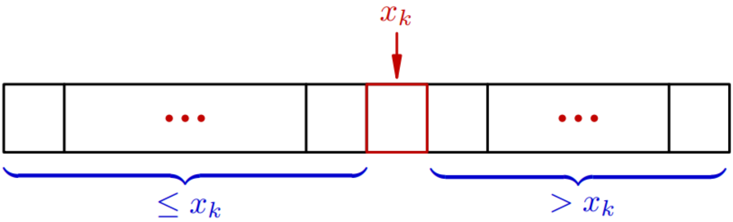
06. ožujka 2023.

QuickSort algoritam

QuickSort algoritam se temelji na principu *podijeli pa vladaj*.

Gruba skica algoritma:

- Uzmemo element x_k niza (nazivamo ga **ključni** element) i dovedemo ga na njegovo **pravo** mjesto u nizu.
- **Lijevo** od x_k stavimo elemente koji su **manji** ili **jednaki** x_k (u proizvoljnom poretku).
- **Desno** od x_k stavimo elemente koji su **veci** od x_k (u proizvoljnom poretku).



QuickSort algoritam

Izbor elementa x_k je **jako bitan** dio algoritma. Postoji više načina na koji se to može napraviti.

Ovisno o izboru x_k:

- Ukoliko se x_k nalazi blizu sredine sortiranog niza, odnosno **prava pozicija** mu je blizu polovice niza, moramo rekurzivno sortirati dva podniza **polovične duljine**.
- U **najgorem** slučaju, x_k se nalazi na rubu sortiranog polja, trebamo sortirati jedan niz duljine $n - 1$.

QuickSort algoritam

Nesortirani dio niza ćemo omeđiti s dva indeksa: lijevim (l) i desnim (d). Ta dva indeksa i polje su argumenti funkcije.

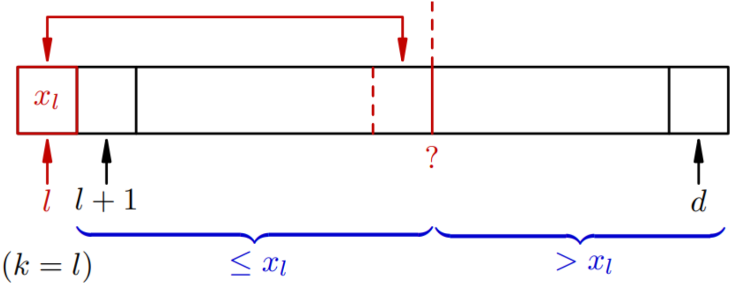
Sortiranje se izvršava ako odabrani dio niza ima barem 2 elementa ($l < d$). Kao **ključni element** se najčešće uzima $k = l$, tj. x_l dovodimo na pravu poziciju u tom komadu niza (poziciju na kojoj treba biti u sortiranom nizu).

Kod takvog izabira nam x_l služi kao granica na lijevom rubu niza.

Dogovor:

- **lijevo**, ispred prave pozicije od x_l stavljamo elemente koji su manji ili jednaki x_l.
- **desno**, iza prave pozicije od x_l stavljamo elemente koji su strogo veći od x_l.

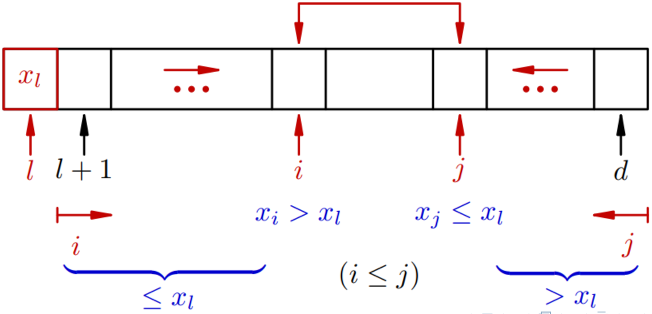
Tada će pravo mjesto elementa x_l biti zadnje u lijevom dijelu.



Prava pozicija elementa x_l se određuje **dvostranim** pretraživanjem po **ostatku** niza.

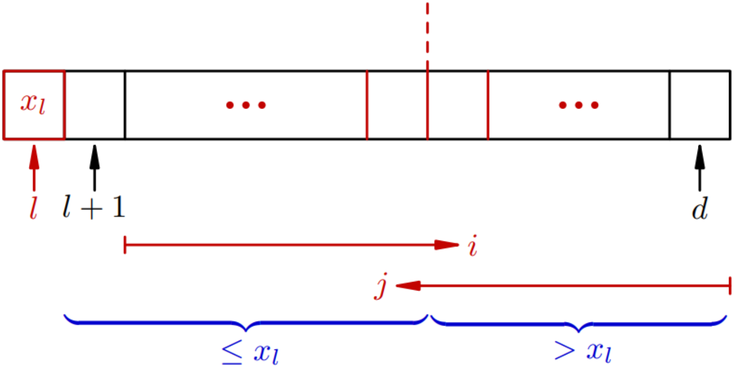
Dva glavna koraka:

- Sa **svake strane** (lijeve i desne) tražimo prvi sljedeći element koji ne pripada toj strani niza.
- Ako nađemo par takvih elemenata, **zamijenimo im mjesta**.



Uvjeti prekida dvostrane pretrage:

- Indeksi i i j moraju biti u **obrnutom** poretku $j < i$.
- **Prava** pozicija elementa x_l je na **indeksu** j (napravimo zamjenu ukoliko je potrebno).



Algoritam za **dvostrano** pretraživanje.

```
if (l < d) {
   i = l + 1;
   j = d;

   /* Prolaz mora i za i == j */
      while (i <= j) {
             while (i <= d && x[i] <= x[l]) ++i;
                 while (x[j] > x[l]) --j;
                 if (i < j) swap(&x[i], &x[j]);
                }
```

Preostali koraci:

- **dovesti** element x_l na njegovu **pravu poziciju** - indeks te pozicije je j.
- **rekurzivno** sortirati **lijevi** i **desni** podniz, bez x_j.

```
if (l < j) swap(&x[j], &x[l]);
quick_sort(x, l, j - 1);
quick_sort(x, j + 1, d);
 }/* Kraj if (l < d). */
```

```c
#include <stdio.h>
/*QuickSort algoritam. x[l] je kljucni element.*/

void swap(int *a, int *b)
{
   int temp;
   temp = *a;
   *a = *b;
   *b = temp;
   return;
}
```

QuickSort algoritam

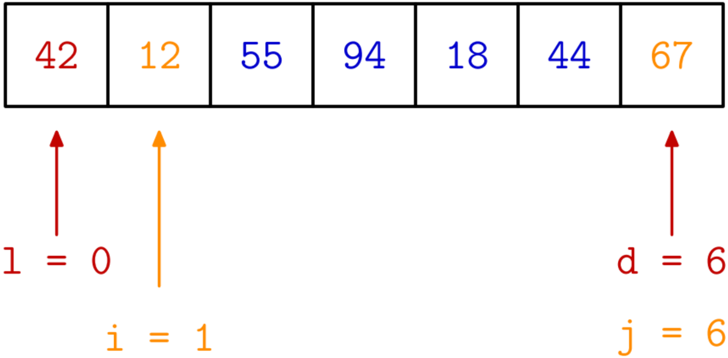
```
void quick_sort(int x[], int l, int d)
{
    int i, j;
    if (l < d) {
        i = l + 1; j = d;

        while (i <= j) {
            while (i <= d && x[i] <= x[l]) ++i;
            while (x[j] > x[l]) --j;
            if (i < j) swap(&x[i], &x[j]); }

      if (l < j) swap(&x[j], &x[l]);
          quick_sort(x, l, j - 1);
          quick_sort(x, j + 1, d);
      }
return; }
```

```c
int main(void) {

    int i, n;
    int x[] = {42, 12, 55, 94, 18, 44, 67};
    n = 7;
    quick_sort(x, 0, n - 1);
    printf("\n Sortirano polje x:\n");
    for (i = 0; i < n; ++i) {
      printf(" x[%d] = %d\n", i, x[i]);
     }

return 0;
}
```

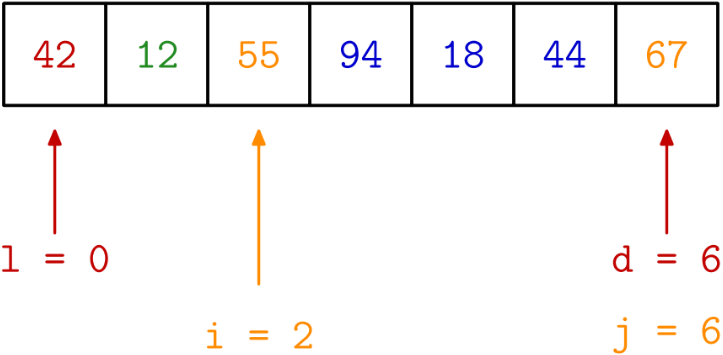
QuickSort algoritam - primjer

QuickSort algoritmom sortiramo polje:

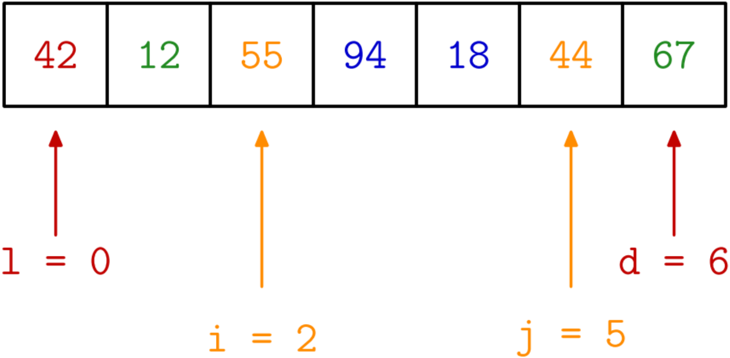
42	12	55	94	18	44	67



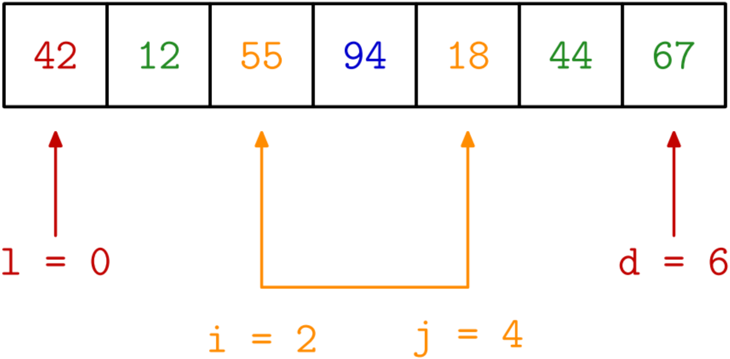
42 je ključni element. Počinjemo dvostranu pretragu. 12 je na dobroj strani ≤ 42.



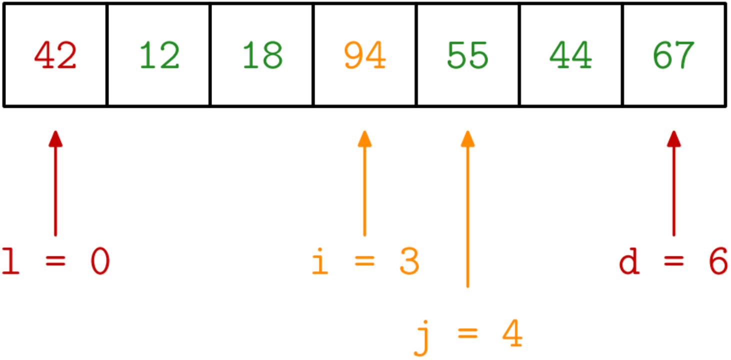
55 je na krivoj strani (> 42), zaustavljamo pretragu s lijeve strane. Pokrećemo pretragu s **desne** strane. 67 je na **dobroj** strani (> 42).



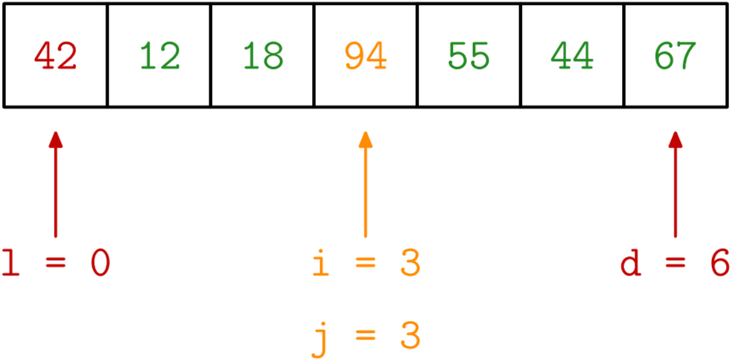
44 je na **dobroj** strani (> 42).



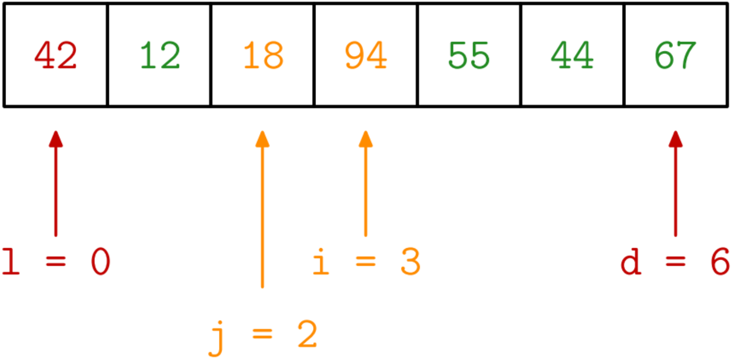
18 je na **krivoj** strani (≤ 42) - stajemo pretragu s desne strane.
$i < j$ - zamjena para elemenata na krivim stranama.



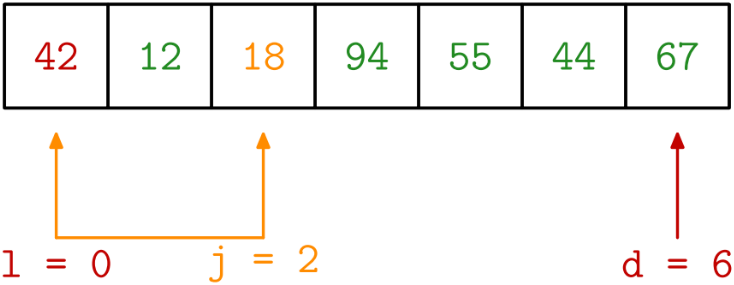
Nastavak dvostrane pretrage s **lijeve** strane. 94 je na krivoj strani ($>$ 42) - zaustavljamo pretragu s lijeve strane.



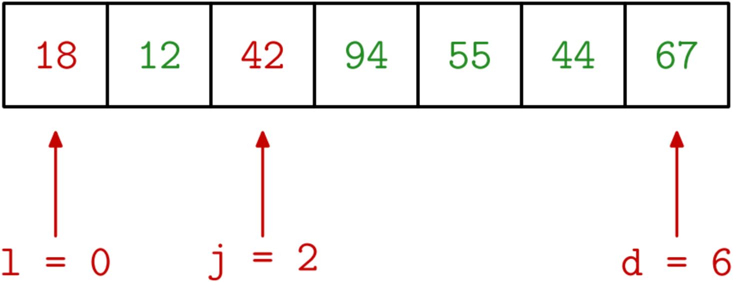
Nastavak dvostrane pretrage s **desne** strane. 94 je na **dobroj** strani (> 42).



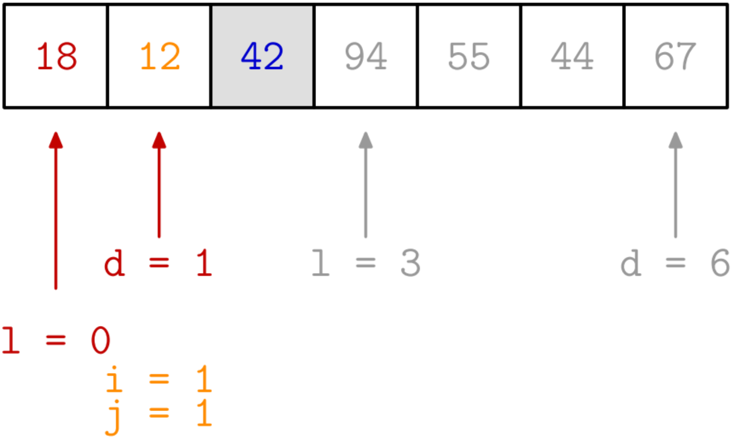
18 je na krivoj strani (≤ 42) - zaustavljamo pretragu s desne strane. $j < i$ - nema zamjene, kraj dvostrane pretrage.



$l < j$, zamjena x_l i x_j.



Prava pozicija za 42 je x_2. Istim postupkom rekurzivno sortiramo podnizove $[x_0,\ x_1]$ i $[x_3,\ x_4,\ x_5,\ x_6]$.



Postupak se nastavlja dok rekurzivno ne dođemo do jednočlanih ili praznih polja, čime je postupak sortiranja završen.

Prosječna složenost algoritma je $\mathcal{O}(n \cdot log_2(n))$ za slučajne **dobro randomizirane** nizove. U najgorem slučaju je složenost $\mathcal{O}(n^2)$ za **već sortirani** i **naopako sortirani** niz.

Algoritam je konstruirao C. A. R. Hoare, 1962. godine.

Poboljšanja QuickSort algoritma

- Za $n = 2, 3$ sortiramo klasičnim algoritmom (provjere zamjena).
- Za $n > 3$ kao ključni element izaberemo *srednji* od neka 3 (ubrzanje oko 30%).
- Kontrola dubine rekurzije (prvo obradi **kraće** od dva preostala polja). Dulje polje smjesti na programski stog.

Uz navedeno, postoji niz drugih optimizacija QuickSort algoritma.

Sortiranje i pretraživanje u standardnoj biblioteci

U standardnoj `C` biblioteci - datoteka zaglavlja `<stdlib.h>`, postoje funkcije:

- `qsort` - QuickSort algoritam za sortiranje niza podataka.
- `bsearch` - Binarno traženje zadanog podatka u sortiranom nizu.

Pri korištenju navedenih funkcija moramo sami zadati funkciju za uspoređivanje podataka u nizu.

Funkcije qsort i bsearch

```
void qsort(void *base, size_t n, size_t size,
int (*comp) (const void *, const void *));

void *bsearch(const void *key, const void *base,
size_t n, size_t size,
int (*comp) (const void *, const void *));
```

Particija (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**.

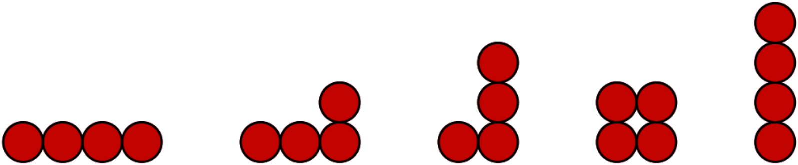
Particija od n ima oblik: $n = a_1 + a_2 + \cdots + a_m$, gdje je $m \in \mathbb{N}$ (broj pribrojnika u rastavu), $a_1, \ldots, a_m$ su **pribrojnici**.

S obzirom na to da poredak pribrojnika nije bitan, možemo smatrati da su pribrojnici poredani (nepadajuće):
$n = a_1 + a_2 + \cdots + a_m, \;\; a_1 \leq a_2 \leq \cdots \leq a_m.$

Zanimljivo je pronaći na koliko (različitih) načina se n može ovako zapisati (rastaviti). **Broj particija** od n označavamo s $p(n)$. Broj pribrojnika m može biti proizvoljan.

Primjer: Napišimo program koji učitava prirodan broj n i ispisuje **broj particija** $p(n) =$ *broj rastava od n* u zbroj nepadajućih prirodnih brojeva.

Primjer za $n = 4$:



$1+1+1+1$ $\quad$ $1+1+2$ $\quad$ $1+3$ $\quad$ $2+2$ $\quad$ 4

Broj particija je: $p(4) = 5$.

Particije

Jedan pristup za rješavanje problema je generirati sve particije od n (u nekom redosljedu) i izbrojati ih.

Kako generirati particije od n?

- generiramo pribrojnik po pribrojnik
- pazimo da pribrojnici **ne padaju** ($a_l \geq a_{l-1}$ za $l \geq 2$).
- pojedini pribrojnik moguće dodati i više puta

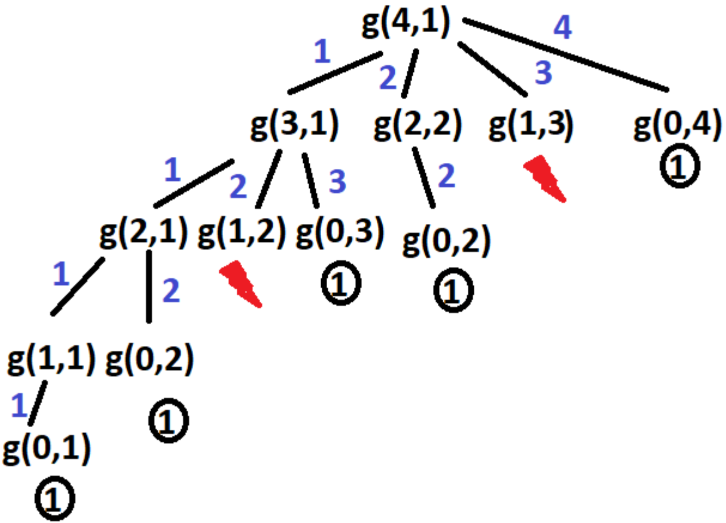
Problem: broj pribrojnika m može varirati od 1 do n. Zbog toga **ne možemo** koristiti niz petlji za rješavanje problema!

Problem možemo riješiti rekurzivno. U svakom pozivu funkcije izvršavamo petlju za izbor pribrojnika a_l. **Rekurzivni pozivi** realiziraju petlje unutar petlji (broj pojavljivanja pribrojnika a_l u sumi).

Označimo s g funkciju koja kao parametre prima trenutni broj n za koji tražimo particiju i broj k koji označava minimalni pribrojnik a_l koji možemo dodati u sljedećem koraku. Funkcija ispituje, na koliko načina možemo particionirati broj n pribrojnicima $a_k \geq a_l$. Nakon dodavanja broja $a_l = k$ u particiju, treba pronaći particiju broja $n - k$ (to je rekurzivni korak). Broj k možemo dodavati maksimalno n puta i brojeve (prema prethodnom razmatranju) dodajemo u uzlaznom poretku.

Terminalni uvjet rekurzije?
Poziv funkcije kod kojeg je $n = 0$ (tada brojač povećamo za 1) ili kada $n < a_l$ (tada ne povećavamo brojač). Ne možemo zapisati broj n kao sumu pribrojnika $> n$.

Funkciju g inicijalno pozovemo: $g(n, 1)$ (na koliko načina možemo particionirati n ukoliko koristimo pribrojnike ≥ 1).


g(4,1)
1
2
3
4
g(3,1)
g(2,2)
g(1,3)
g(0,4)
1
1
2
3
2
g(2,1)
g(1,2)
g(0,3)
g(0,2)
1
1
1
2
g(1,1)
g(0,2)
1
1
g(0,1)
1

Particije

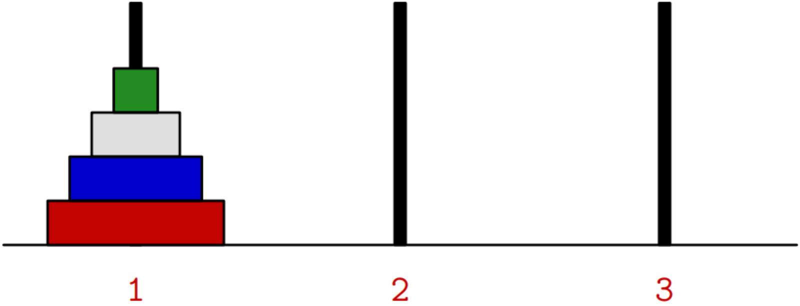
```c
#include <stdio.h>

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

      if (suma == 0) return 1;

      for (i = prvi; i <= suma; ++i)
            broj += particije(suma - i, i);
return broj;
}
```

```
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;
}
```

Trag poziva dobijemo 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;
}
```

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



Pravila igre:

- u svakom potezu se može prebaciti samo jedan disk (najgornji na nekom štapu, 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.

Prebacujemo jedan po jedan disk tako da stavljamo samo manji disk na veći ili na prazan štap.

Hanojski tornjevi

Osnovni korak u ovom problemu je prebacivanje najgornjeg (najmanjeg) diska s jednog štapa (**odakle**) na drugi (**kamo**).

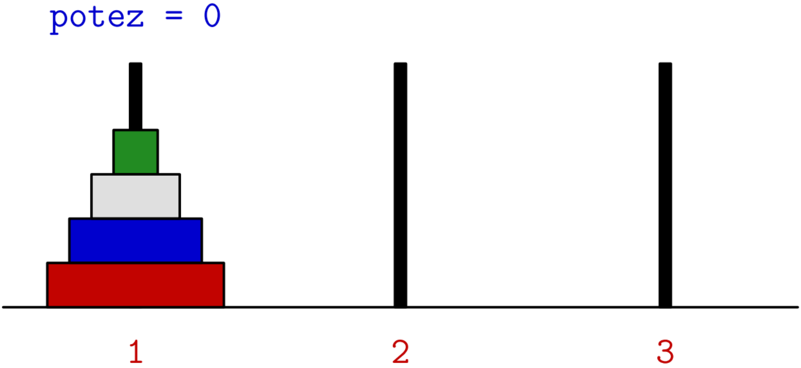
Ideja: svesti problem za n diskova na **isti** problem s **manje diskova**. Da bi to postigli, koristit ćemo **osnovni potez** za neki disk (ili diskove).

Želimo rekurzivno smanjivati n dok ne bude $n = 1$. Taj slučaj riješimo korištenjem **osnovnog poteza**.

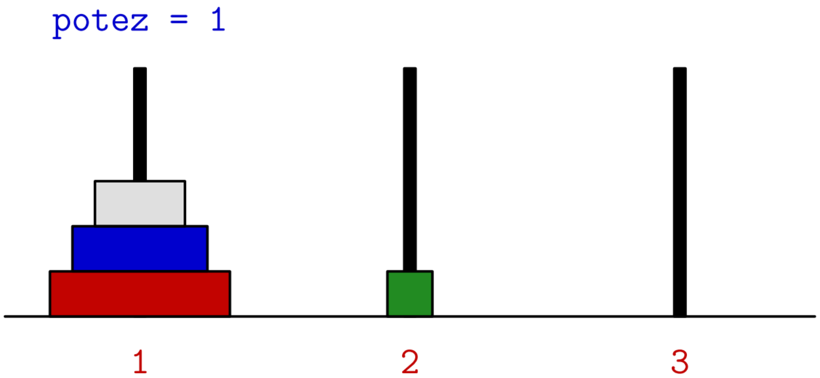
- Tražimo **parametrizaciju** rekurzije kojom bi mogli svesti problem na **isti** problem s **manje diskova**.
- **Štapove** treba uključiti u **parametrizaciju**, uloge štapova **odakle** i **kamo** će morati **varirati**.
- Problem prebacivanja n diskova s **odakle** na **kamo** pišemo kao poziv funkcije `prebaci(n, odakle, kamo)`.

Ključno pitanje: kojih n diskova treba uzeti za **rekurzivna** prebacivanja a na **koji** disk treba primijeniti **osnovni** potez?

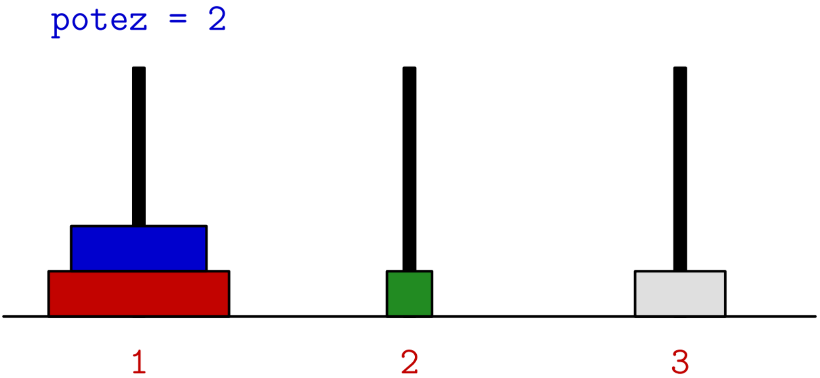
- Rekurzivna funkcija `prebaci` **prebacuje** najgornjih $n-1$ diskova na polaznom štapu s **odakle** na pomoćni štap.
- Najdonji disk je najveći pa ne blokira prebacivanje gornjih diskova.
- Taj disk osnovnim korakom prebacujemo na odredišni štap.
- Gornjih $n-1$ diskova prebacujemo s pomoćnog na odredišni štap.


potez = 0
1
2
3

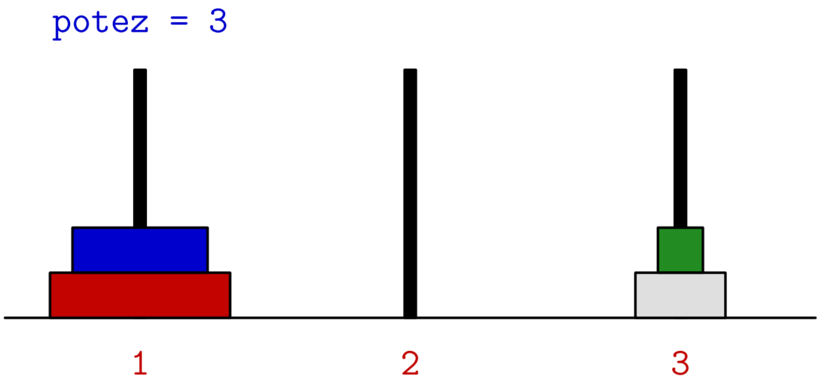
Hanojski tornjevi - primjer

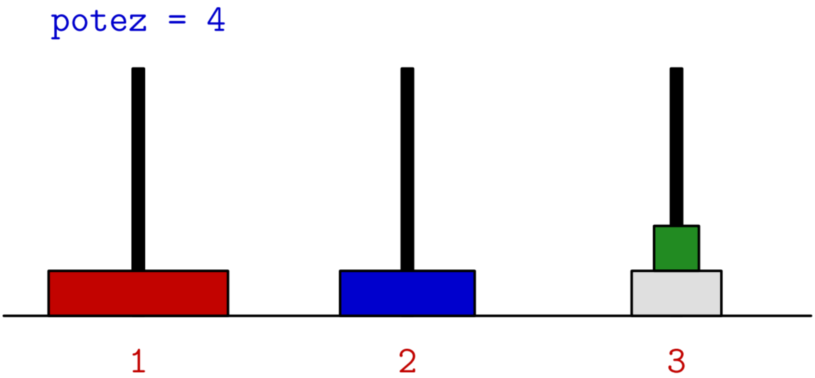


Hanojski tornjevi - primjer

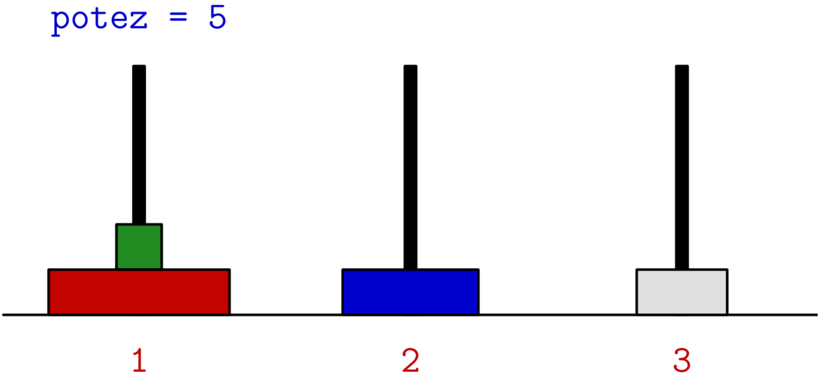


Hanojski tornjevi - primjer

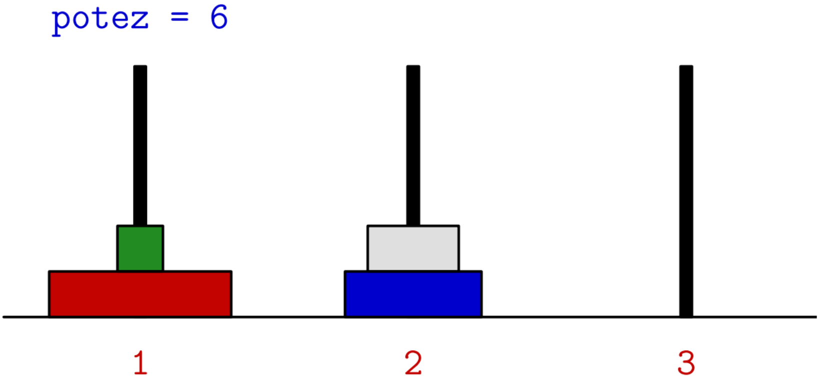



potez = 4
1
2
3

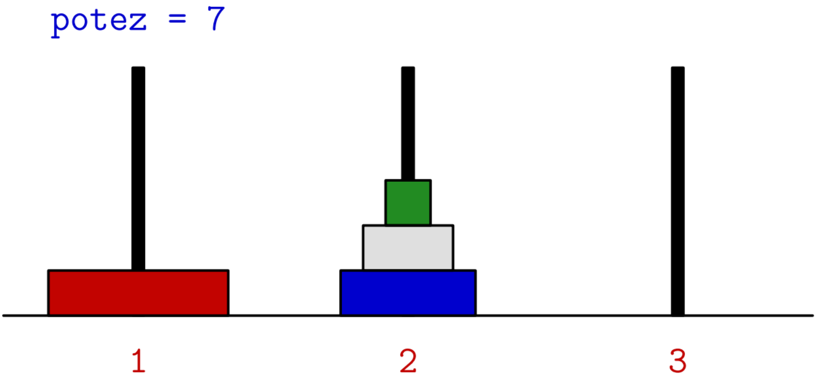
Hanojski tornjevi - primjer



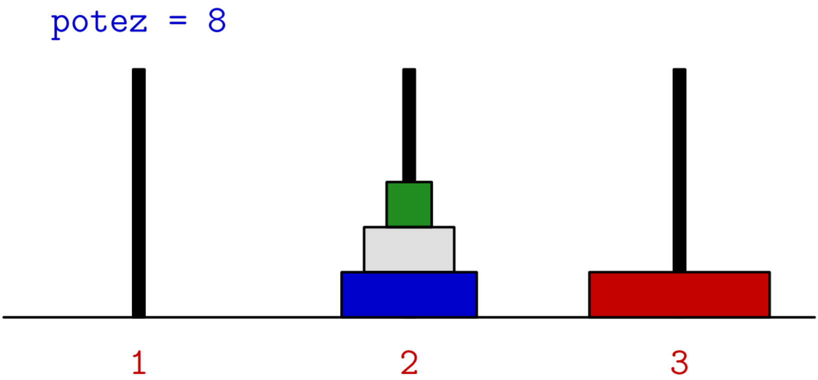
Hanojski tornjevi - primjer



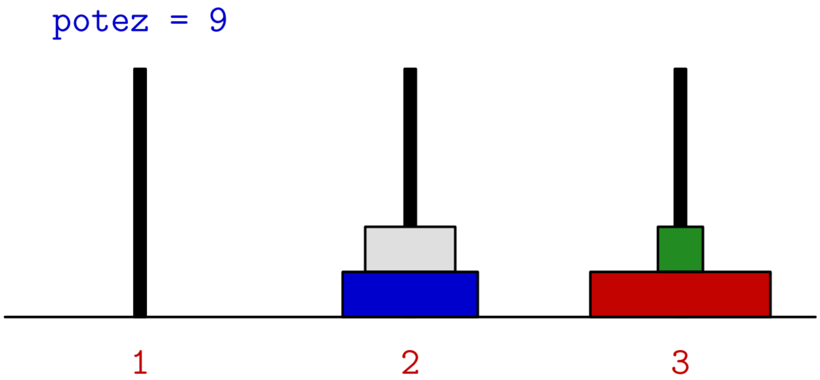
Hanojski tornjevi - primjer



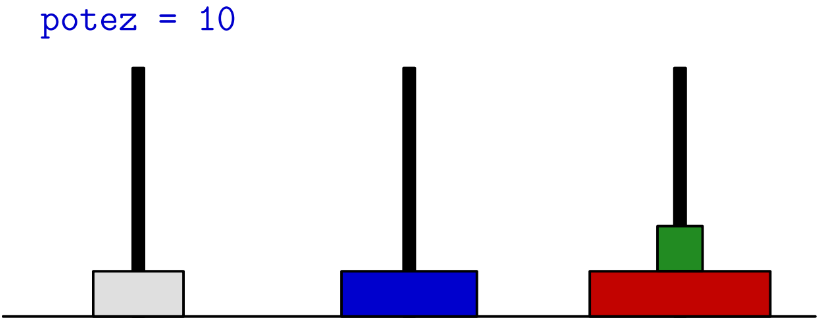
Hanojski tornjevi - primjer



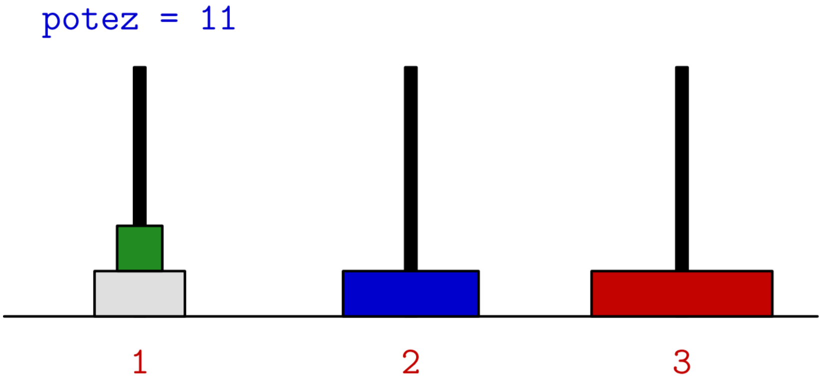
Hanojski tornjevi - primjer



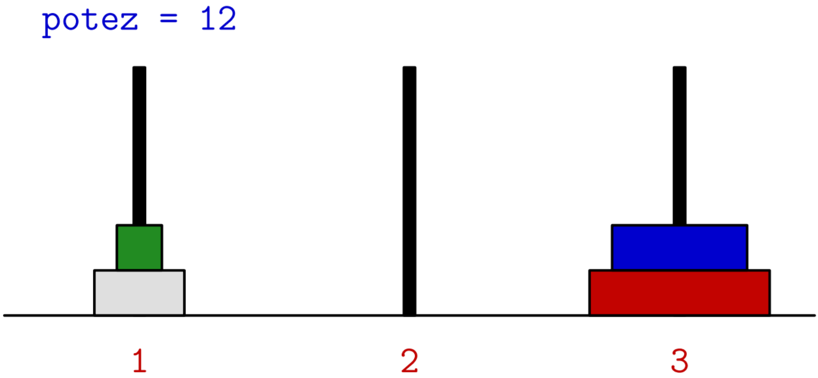
Hanojski tornjevi - primjer



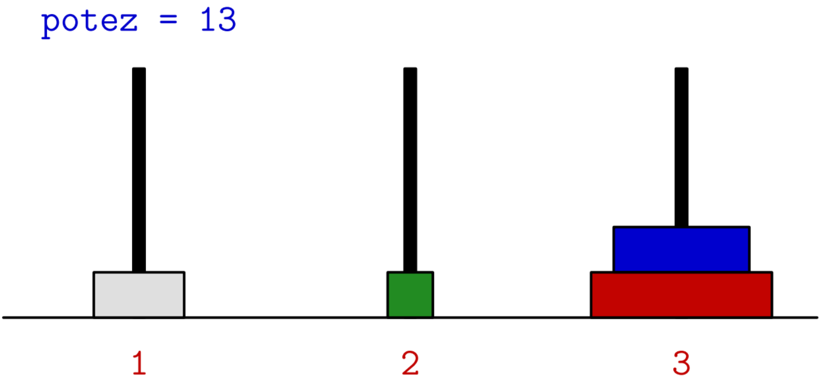
Hanojski tornjevi - primjer



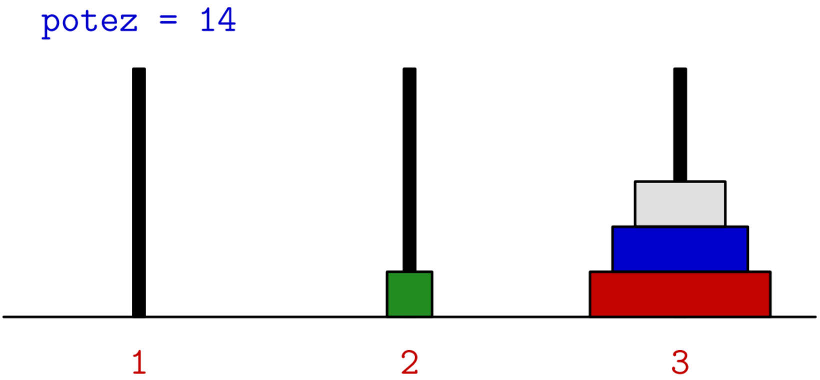
Hanojski tornjevi - primjer



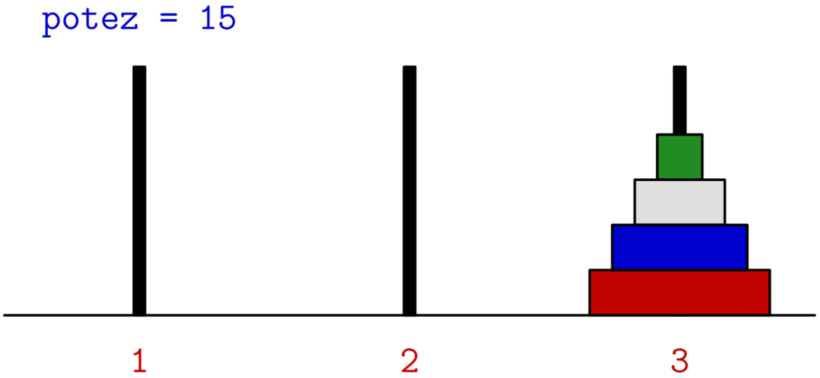
Hanojski tornjevi - primjer



Hanojski tornjevi - primjer



Hanojski tornjevi - primjer



Animaciju Hanojskih tornjeva možete naći na web stranici:
`https://www.mathsisfun.com/games/towerofhanoi.html`

Program za rješavanje problema **Hanojskih tornjeva** ima dvije funkcije:

- `prebaci_jednog` - realizira **osnovni potez** prebacivanja **jednog** (najgornjeg) s jednog štapa na drugi.
- `Hanojski_tornjevi` - realizira **rekurzivno** prebacivanje **gornjih** n diskova u obliku:
  - prebaci **gornjih** $n-1$ diskova s pomoćnog na odredišni štap.
  - prebaci **jedan** disk (najdonji) na odredišni štap.
  - prebaci gornjih $n-1$ disk s pomoćnog štapa na odredišni štap.

Precizna realizacija funkcija ovisi o tome što želimo dobiti kao **rješenje problema** (redoslijed koraka ili broj osnovnih poteza).

Računamo redoslijed koraka (poteza).

- `prebaci_jednog` mora znati **odakle** i **kamo** prebacuje
- `Hanojski_tornjevi` osim *n* mora sadržavati informaciju **odakle** i **kamo** prebacuje. Zbog jednostavnosti dodajemo informaciju i o pomoćnom štapu (može se izračunati iz preostala dva).

```
#include <stdio.h>
void prebaci_jednog(int odakle, int kamo){
   printf(" prebaci s %d na %d\n", odakle, kamo);
return;
}//ispisuje osnovni potez
```

Hanojski tornjevi - program

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

Hanojski tornjevi - program

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

for (n = 1; n <= 5; ++n) {
  printf("\n Prebaci %d diskova s 1 na 3:\n", n);
  Hanoj(n, 1, 3, 2);
}//Broj diskova iteriramo od 1 do 5
return 0;
}
```

Hanojski tornjevi - broj poteza

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

$$h_n = h_{n-1} + 1 + h_{n-1} = 2h_{n-1} + 1, \;\; n \geq 1$$

Rješenje gornje nehomogene rekurzije je: $h_n = 2^n - 1, \;\; n \geq 0$ (način rješavanja na višim godinama studija).

Gornji algoritam pokazuje:

- polazni problem za n diskova uvijek ima rješenje za svaki n,
- minimalni broj poteza je jednak $2^n - 1$

optimalno rješenje je **jedinstveno**.

Razlog brzog rasta broja poteza je činjenica da koristimo samo 3 štapa.

Poopćenje problema dobijemo korištenjem $k \geq 3$ štapova. Tada problem ima više rješenja a traži se minimalan broj poteza.

Ograničeni problem Hanojskih tornjeva: imamo 3 štapa uz dodatno ograničenje:

- u svakom potezu se disk smije prebaciti s nekog štapa samo na **susjedni** štap.
- **zabranjeno** je izravno prebacivanje s **prvog** na **treći** štap ili obratno (minimalno $3^n - 1$ koraka).