Algoritmi u kombinatorici

https://arena.petlja.org/competition/mgkombnatorika2019

1.

vreme	memorija
1 s	64 Mb

Napiši program koji određuje sve varijacije sa ponavljanjem dužine k skupa 1,…,n.
Ulaz: Sa standardnog ulaza se učitava broj n (1≤n≤5) i u narednoj liniji broj k (1≤k≤5).

Izlaz: Na standardni izlaz ispisati sve varijacije, sortirane leksikografski.

Primer

Ulaz

```
2
3
```

Izlaz

```
1 1 1
1 1 2
1 2 1
1 2 2
2 1 1
2 1 2
2 2 1
2 2 2
```

Rešenje

Rekurzivno generisanje varijacija

Varijacije se mogu nabrojati induktivno rekurzivnom konstrukcijom.

BAZA

Jedina varijacija dužine nula je prazna.

INDUKTIVNI KORAK

Sve varijacije dužine k se mogu dobiti tako što se na prvo mesto upiše bilo koji od brojeva od 1 do n, a zatim se preostala mesta dopune svim varijacijama dužine k−1.

Implementaciju ćemo organizovati tako da umesto da vraća kolekciju varijacija, rekurzivna funkcija prima delimično popunjen niz koji će na sve moguće načine dopunjavati varijacijama tekuće dužine k (koja će se smanjivati kroz rekurzivne pozive).

Dakle, na tekuću poziciju u nizu (ona se izračunava kao razlika između dužine niza i trenutne vrednosti k) postavljamo jednu po jednu vrednost od 1 do n i zatim rekurzivno pozivamo funkciju da popuni ostatak niza (time što smanjujemo dužinu k i time prelazimo na narednu poziciju).

Recimo i da je moguće na poslednje mesto postavljati jedan po jedan broj od 1 do n, a zatim rekurzivno popunjavati prefiks, no time bi redosled varijacija bio drugačiji od traženog.

```
#include <iostream>
#include <vector>

using namespace std;

// ispisuje tekucu varijaciju na standardni izlaz
void obradi(const vector<int>& varijacija) {
  for (int x : varijacija)
    cout << x << " ";
  cout << endl;
}

// sve varijacije duzine k elemenata skupa {1, ..., n}
// Dati niz varijacija duzine varijacije.size() - k
// se dopunjuje svim mogucim varijacijama sa ponavljanjem
// duzine k skupa {1, ..., n} i sve tako
// dobijene varijacije se obrađuju
void obradiSveVarijacije(int k, int n,
                         vector<int>& varijacija) {
  // k je 0, pa je jedina varijacija duzine nula prazna i njenim
  // dodavanjem na polazni niz on se ne menja
  if (k == 0)
    obradi(varijacija);
  else
    // na tekucu poziciju postavljamo sve moguce vrednosti od 1 do n i
    // dobijeni niz onda rekurzivno proširujemo
    for (int nn = 1; nn <= n; nn++) {
      varijacija[varijacija.size() - k] = nn;
      obradiSveVarijacije(k-1, n, varijacija);
    }
}


// sve varijacije duzine k skupa {1, ..., n}
void obradiSveVarijacije(int k, int n) {
  vector<int> varijacija(k);
  obradiSveVarijacije(k, n, varijacija);
}


int main() {
  int n, k;
  cin >> n >> k;
  obradiSveVarijacije(k, n);
  return 0;
}
```

Pronalaženje leksikografski sledeće varijacije

Druga mogućnost je da se krene od leksikografski najmanje varijacije (to je varijacija sa k jedinica 11…11) i da se korišćenjem funkcije sledeća varijacija određuje naredna varijacija date varijacije u odnosu na leksikografski redosled, sve dok takva postoji.

```
#include <iostream>
#include <vector>

using namespace std;

// ispisuje tekucu varijaciju na standardni izlaz
void obradi(const vector<int>& varijacija) {
  for (int x : varijacija)
    cout << x << " ";
  cout << endl;
}

bool sledecaVarijacija(int n,
                       vector<int>& varijacija) {
   // od kraja varijacije tražimo prvi element koji se moze povecati
   int i;
   int k = varijacija.size();
   for (i = k-1; i >= 0 && varijacija[i] == n; i--)
      varijacija[i] = 1;
   // svi elementi su jednaki n - ne postoji naredna varijacija
   if (i < 0) return false;
   // uvecavamo element koji je moguće uvecati
   varijacija[i]++;
   return true;
}

void obradiSveVarijacije(int k, int n) {
   // krećemo od varijacije 11...11 - ona je leksikografski najmanja
   vector<int> varijacija(k, 1);
   // obradjujemo redom varijacije dok god postoji leksikografski
   // sledeca
   do {
     obradi(varijacija);
   } while(sledecaVarijacija(n, varijacija));
}

int main() {
  int n, k;
  cin >> n >> k;
  obradiSveVarijacije(k, n);
  return 0;
}
```

2. Kombinacije

vreme	memorija
1 s	64 Mb

Kombinacije dužine k od n elemenata podrazumevaju da se vrši odabir k elemenata skupa {1,…,n}, slično kao što se, na primer, u igri loto bira 7 od 39 kuglica. Napiši program koji za date vrednosti k i n nabraja i ispisuje sve kombinacije, poređane po leksikografskom redosledu.

Ulaz

Prva linija standardnog ulaza sadrži broj k (1≤k≤n), a naredna broj n (2≤n≤20).

Izlaz

Na standardni izlaz ispisati sve kombinacije. Svaka kombinacija treba da bude predstavljena nizom brojeva sortiranim strogo rastuće, a sve kombinacije treba da budu poređane u leksikografskom redosledu.

Primer

Ulaz

```
3
5
```

Izlaz

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

Rekurzivni pozivi po pozicijama

Zadatak rekurzivne funkcije biće da dopuni niz dužine k od pozicije i pa do kraja. Kada je i=k, niz je popunjen i potrebno je obraditi dobijenu kombinaciju. U suprotnom biramo element koji ćemo postaviti na poziciju i. Pošto su kombinacije uređene strogo rastuće, on mora biti veći od prethodnog (ako prethodni ne postoji, onda može biti 1 i manji ili jednak n.

Zapravo, ovo gornje ograničenje mora da se smanji. Pošto su elementi strogo rastući, a od pozicije i pa do kraja niza treba postaviti k−i elemenata, onda na poziciji i može biti n+i−k+1 i tada će na poziciji k−1 biti vrednost n.

U petlji stavljamo jedan po jedan od tih elemenata na poziciju i i rekurzivno nastavljamo generisanje od naredne pozicije.

```
#include <iostream>
#include <vector>

using namespace std;

// ispisuje kombinaciju na standarndi izlaz
void obradi(const vector<int>& kombinacija) {
  for (int x : kombinacija)
    cout << x << " ";
  cout << endl;
}

// niz kombinacije dužine k na pozicijama [0, i) sadrži uređen
// niz elemenata iz skupa [1, n-i+1). Procedura na sve moguće
// načine dopunjava elementima iz skupa [1, n) tako da niz bude
// uređen rastući
void obradiSveKombinacije(vector<int>& kombinacija, int i, int n) {
  // tražena dužina kombinacije
  int k = kombinacija.size();

  // ako je popunjen ceo niz samo ispisujemo kombinaciju
  if (i == k) {
    obradi(kombinacija);
    return;
  }
  // određujemo raspon elemenata na poziciji i
  int pocetak = i == 0 ? 1 : kombinacija[i-1]+1;
  int kraj = n + i - k + 1;
  // jedan po jedan element upisujemo na poziciju i, pa
  // nastavljamo generisanje rekurzivno
  for (int x = pocetak; x <= kraj; x++) {
    kombinacija[i] = x;
    obradiSveKombinacije(kombinacija, i+1, n);
  }
}

// nabraja i obradjuje sve kombinacije dužine k skupa
// {1, 2, ..., n}
void obradiSveKombinacije(int k, int n) {
  vector<int> kombinacija(k);
  obradiSveKombinacije(kombinacija, 0, n);
}

int main() {
  int k, n;
  cin >> k >> n;
  obradiSveKombinacije(k, n);
  return 0;
}
```

Rekurzivni pozivi po vrednostima

Postoji način da izbegnemo rekurzivne pozive u petlji. Tokom rekurzije možemo da čuvamo informaciju o tome koji je raspon elemenata kojim se proširuje niz. Znamo da su to elementi skupa 1,…,n, međutim, pošto su kombinacije sortirane rastuće skup kandidata je uži. U prethodnom programu smo najmanju vrednost za poziciju i određivali na osnovu vrednosti sa pozicije i−1.
ALI, alternativno možemo i eksplicitno da održavamo promenljive nmin i nmax koje određuju skup nmin,…,nmax čiji se elementi raspoređuju u kombinaciji na pozicijama iz intervala [i,k). Ako je taj interval prazan, kombinacija je popunjena i može se obraditi. U suprotnom, ako je n*{min}* >*n*{max},
tada ne postoji vrednost koju je moguće staviti na poziciju, i,,te možemo izaći iz rekurzije, jer se trenutna kombinacija ne može popuniti do kraja.

```
#include <iostream>
#include <vector>

using namespace std;

// ispisuje kombinaciju na standarndi izlaz
void obradi(const vector<int>& kombinacija) {
  for (int x : kombinacija)
    cout << x << " ";
  cout << endl;
}

void obradiSveKombinacije(vector<int>& kombinacija, int i,
                          int n_min, int n_max) {
  // tražena dužina kombinacije
  int k = kombinacija.size();

  // ako je popunjen ceo niz samo ispisujemo kombinaciju
  if (i == k) {
    obradi(kombinacija);
    return;
  }

  // ako tekuću kombinaciju nije moguće popuniti do kraja
  // prekidamo ovaj pokušaj
  if (n_max - n_min + 1 < k - i)
    return;

  // vrednost n_min uključujemo na poziciju i, pa rekurzivno
  // proširujemo tako dobijenu kombinaciju
  kombinacija[i] = n_min;
  obradiSveKombinacije(kombinacija, i+1, n_min+1, n_max);
  // vrednost n_min preskačemo i isključujemo iz kombinacije
  obradiSveKombinacije(kombinacija, i, n_min+1, n_max);
}
```

```
// nabraja i obradjuje sve kombinacije dužine k skupa
// {1, 2, ..., n}
void obradiSveKombinacije(int k, int n) {
  vector<int> kombinacija(k);
  obradiSveKombinacije(kombinacija, 0, 1, n);
}

int main() {
  int k, n;
  cin >> k >> n;
  obradiSveKombinacije(k, n);
  return 0;
}
```

Leksikografski sledeća kombinacija

Jedan način da se zadatak reši bez rekurzije je da se upotrebi funkcija za određivanje naredne kombinacije u leksikografskom poretku koja je opisana u potprogramu sledeca Kombinacija.

```
#include <iostream>
#include <vector>

using namespace std;

// ispisuje kombinaciju na standarndi izlaz
void obradi(const vector<int>& kombinacija) {
  for (int x : kombinacija)
    cout << x << " ";
  cout << endl;
}

// pronalazi sledecu kombinaciju u leksikografskom redosledu
bool sledecaKombinacija(int n, vector<int>& kombinacija) {
  // tražena dužina kombinacije
  int k = kombinacija.size();

  // krećemo od kraja i tražimo prvu poziciju koja nije na maksimumu
  // tj. koja se može povećati. Maksimumi od kraja su n, n-1, n-2, ...
  int i;
  for (i = k-1; i >= 0 && kombinacija[i] == n; i--, n--)
    ;
  // ako takva pozicija ne postoji, tekuća kombinacija je maksimalna
  if (i < 0)
    return false;
  // uvećavamo poslednji element koji se može povećati
  kombinacija[i]++;
  // iza njega slažemo redom brojeve za jedan veće
  for (i++; i < k; i++)
    kombinacija[i] = kombinacija[i-1] + 1;
  return true;
}


// nabraja i obradjuje sve kombinacije dužine k skupa
// {1, 2, ..., n}
```

```
void obradiSveKombinacije(int k, int n) {
  // krecemo od kombinacije 1, 2, ..., k
  vector<int> kombinacija(k);
  for (int i = 0; i < k; i++)
    kombinacija[i] = i + 1;

  // obradjujemo kombinacije dokle god postoji sledeca
  do {
    obradi(kombinacija);
  } while (sledecaKombinacija(n, kombinacija));
}

int main() {
  int k, n;
  cin >> k >> n;
  obradiSveKombinacije(k, n);
  return 0;
}
```

Leksikografski sledeća kombinacija - optimizovano rešenje

Postoji i mala optimizacija prethodnog postupka. Naime, ako znamo vrednost prelomne tačke u jednom koraku, bez ponovne pretrage možemo odrediti vrednost prelomne tačke u narednom koraku. Ključno pitanje je to da li nakon uvećanja prelomne vrednosti ona dostiže svoj maksimum.

- Ako dostiže tj. ako posle uvećanja važi $komb_i=n-k+i+1$, tada posle uvećanja prelomne vrednosti, redom iza prelomne tačke moraju naći elementi koji su svi na svojim maksimumima, međutim, to je već slučaj tako da nije potrebno ponovo ih ažurirati. Naredna prelomna tačka je neposredno ispred tekuće prelomne tačke. Na primer, naredna kombinacija za 1356 je 1456.
  Prelomna tačka je i=1, važi da je $komb_1=3=6-4+1$ i dovoljno je samo uvećati element 3 na 4. Pošto su elementi od 4, 5 i 6, na svojoj maksimalnoj vrednosti, znamo da je naredna prelomna vrednost 1, pa je naredna kombinacija 2345.
- Ako nakon uvećanja prelomna vrednost ona ne dostiže svoj maksimum tj. ako nakon uvećanja važi $komb_i<n-k+i+1$, onda je nakon uvećanja i popunjavanja niza do kraja poslednji element sigurno ispod svoje maksimalne vrednosti, tako da je naredna prelomna tačka poslednja pozicija u nizu.
  Interesantno, ova "optimizacija" ne donosi nikakvu značajnu dobit i nema pratkičnih implikacija. Ako obrada uključuje bilo kakvu netrivijalnu operaciju ili ispis kombinacije na ekran, obrada potpuno dominira vremenom generisanja. Ako je obrada trivijalna (na primer, samo uvećanje globalnog brojača za jedan) tada ne postoji značajna razlika u vremenu izvršavanja. Razlog tome je to što je u većini slučajeva prelomna tačka poslednji element ili je vrlo blizu desnog kraja, pa je linearna pretraga brza.

```
#include <iostream>
#include <vector>

using namespace std;
```

```
// ispisuje kombinaciju na standarndi izlaz
void obradi(const vector<int>& kombinacija) {
  for (int x : kombinacija)
    cout << x << " ";
  cout << endl;
}

// nabraja i obradjuje sve kombinacije dužine k skupa
// {1, 2, ..., n}
void obradiSveKombinacije(int k, int n) {
     // krecemo od kombinacije 1, 2, ..., k
   vector<int> kombinacija(k);
   for (int i = 0; i < k; i++)
     kombinacija[i] = i + 1;

   // obrađujemo prvu kombinaciju
   obradi(kombinacija);

   // specijalno obrađujemo slučaj n = k kada
   // nema drugih kombinacija
   if (n == k) return;

   // prva prelomna tačka je poslednja pozicija u nizu
   int i = k-1;
   while (i >= 0) {
      // ažuriramo kombinaciju
      kombinacija[i]++;
      // ako je uvećani prelomni element dostigao maksimum
      if (kombinacija[i] == n - k + i + 1)
        // naredna prelomna tačka je neposredno pre njega
        i--;
      else {
        // popunjavamo niz do kraja
        for (int j = i+1; j < k; j++)
           kombinacija[j] = kombinacija[j-1] + 1;
        // naredna prelomna tačka je poslednji element niza
        i = k-1;
      }
      // obrađujemo dobijenu kombinaciju
      obradi(kombinacija);
   }
}

int main() {
  int k, n;
  cin >> k >> n;
  obradiSveKombinacije(k, n);
  return 0;
}
```

3. Permutacije

vreme	memorija	ulaz	izlaz
1 s	64 Mb	standardni ulaz	standardni izlaz

Napiši program koji generiše i ispisuje sve permutacije skupa {1,2,…,n}.

Ulaz Sa standardnog ulaza se učitava broj n (1≤n≤8).

Izlaz Na standardni izlaz ispisati tražene permutacije. Svaku permutaciju ispisati u posebnom redu, a elemente razdvojiti po jednim razmakom. Redosled permutacija može biti proizvoljan.

Primer

Ulaz

```
3
```

Izlaz

```
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

Rešenje

Rekurzivno generisanje permutacija

Rekurzivno generisanje permutacija u leksikografskom redosledu je veoma komplikovano, tako da ćemo se odreći uslova da permutacije moraju biti poređane leksikografski.
U tom slučaju možemo postupiti na sledeći način. Na prvu poziciju u nizu u kojem čuvamo tekuću permutaciju treba da postavljamo jedan po jedan element skupa, a zatim da rekurzivno određujemo sve permutacije preostalih elemenata. Fiksirane elemente i elemente koje treba permutovati možemo čuvati u istom nizu. Neka na pozicijama [0,k) čuvamo elemente koje treba permutovati, a na pozicijama [k,n) čuvamo fiksirane elemente.
Razmatramo poziciju k−1. Ako je k=1, tada postoji samo jedna permutacija jednočlanog niza na poziciji 0, nju pridružujemo fiksiranim elementima (pošto je ona već na mestu 0 nema potrebe ništa dodatno raditi) i ispisujemo permutaciju. Ako nijje k jednako 1,tada je situacija komplikovanija.
Jedan po jedan element dela niza sa pozicija [0,k)treba da dovodimo na mesto k-1.

Jedan po jedan element dela niza sa pozicija [0,k) treba da dovodimo na mesto k-1 i da rekurzivno pozivamo permutovanje dela niza na pozicijama [0, k-1).
Ideja koja se prirodno javlja je da vršimo razmenu elementa na poziciji k-1 redom sa svim elementima iz intervala [0, k) i da nakon svake razmene vršimo rekurzivne pozive.
Na primer,
Ako je niz na početku 123,onda menjamo element,3 sa elementom 1, dobijamo 321 i pozivamo rekurzivno generisanje permutacija niza 32 sa fiksiranim elementom 1 na kraju. Zatim u početnom nizu menjamo element na kraju.

Međutim, sa tim pristupom može biti problema. Naime, da bismo bili sigurni da će na poslednju poziciju stizati svi elementi niza, razmene moramo da vršimo u odnosu na *početno* stanje niza. Jedan način je da se pre svakog rekurzivnog poziva pravi kopija niza, ali postoji i efikasnije rešenje. Naime, možemo kao invarijantu funkcije nametnuti da je nakon svakog rekurzivnog poziva raspored elemenata u nizu isti kao pre poziva funkcije. Ujedno to treba da bude i invarijanta petlje u kojoj se vrše razmene. Na ulasku u petlju raspored elemenata u nizu biće isti kao na ulasku u funkciju. Vršimo prvu razmenu, rekurzivno pozivamo funkciju i na osnovu invarijante rekurzivne funkcije znamo da će raspored nakon rekurzivnog poziva biti isti kao pre njega. Da bismo održali invarijantu petlje, potrebno je niz vratiti u početno stanje. Međutim, znamo da je niz promenjen samo jednom razmenom, tako da je dovoljno uraditi istu tu razmenu i niz će biti vraćen u početno stanje. Time je invarijanta petlje očuvana i može se preći na sledeću poziciju. Kada se petlja završi, na osnovu invarijante petlje znaćemo da je niz isti kao na ulazu u funkciju. Na osnovu toga znamo i da će invarijanta funkcije biti održana i nije potrebno uraditi ništa dodatno nakon petlje.

```
#include <iostream>
#include <vector>

using namespace std;

// ispisuje permutaciju na standarndi izlaz
void obradi(const vector<int>& permutacija) {
  for (int x : permutacija)
    cout << x << " ";
  cout << endl;
}

void obradiSvePermutacije(vector<int>& permutacija, int k) {
  if (k == 1)
    obradi(permutacija);
  else {
    for (int i = 0; i < k; i++) {
      swap(permutacija[i], permutacija[k-1]);
      obradiSvePermutacije(permutacija, k-1);
      swap(permutacija[i], permutacija[k-1]);
    }
```

```
  }
}

void obradiSvePermutacije(int n) {
  vector<int> permutacija(n);
  for (int i = 1; i <= n; i++)
    permutacija[i-1] = i;
  obradiSvePermutacije(permutacija, n);
}

int main() {
  int n;
  cin >> n;
  obradiSvePermutacije(n);
  return 0;
}
```

Određivanje sledeće permutacije

Sve permutacije u leksikografskom redosledu se mogu dobiti tako što se krene od početne permutacije 1,2,…,n i u svakom koraku se ispisuje tekuća permutacija i menja se sa narednom permutacijom u leksikografskom poretku.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

bool sledecaPermutacija(vector<int>& a){
  int n = a.size();

  // linearnom pretragom pronalazimo prvu poziciju i takvu da
  // je a[i] > a[i+1]
  int i = n - 2;
  while (i >= 0 && a[i] > a[i+1])
    i--;
  // ako takve pozicije nema, permutacija a je leksikografski maksimalna
  if (i < 0) return false;
  // linearnom pretragom pronalazimo prvu poziciju j takvu da
  // je a[j] > a[i]
  int j = n - 1;
  while (a[j] < a[i])
    j--;
  // razmenjujemo elemente na pozicijama i i j
  swap(a[i], a[j]);
  // obrcemo deo niza od pozicije i+1 do kraja
  for (j = n - 1, i++; i < j; i++, j--)
    swap(a[i], a[j]);
  return true;
}

void obradiSvePermutacije(int n) {
  vector<int> permutacija(n);
  for (int i = 0; i < n; i++)
    permutacija[i] = i + 1;
```

```
  do {
    obradi(permutacija);
  } while (sledecaPermutacija(permutacija));
}

int main() {
  int n;
  cin >> n;
  obradiSvePermutacije(n);
}
```

Bibliotečka funkcija za sledeću permutaciju

U jeziku C++ funkcija next_permutation deklarisana u zaglavlju određuje narednu permutaciju u odnosu na datu. Funkciji se prosleđuju dva iteratora koji ograničavaju raspon elemenata u kojima se nalazi permutacija.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>

using namespace std;

int main() {
  int n;
  cin >> n;
  vector<int> permutacija(n);
  iota(begin(permutacija), end(permutacija), 1);
  do {
    for (int x : permutacija)
      cout << x << " ";
    cout << endl;
  } while (next_permutation(begin(permutacija), end(permutacija)));
}
```

Heap-ov algoritam

Ovaj algoritam generiše sve moguće permutacije niza od n članova. Algoritam je kreirao B. R. Heap, 1963. godine. Strategija je da se minimizuje pomeranje elemenata u odnosu na naivni algoritam koji smo prvi izložili i koji ima dosta razmena. Osnovna ideja je da se generiše svaka permutacija na osnovu prethodne preko razmene jednog para elemenata; preostalih $n-2$ elemenata se ne premeštaju.

Pri naivnoj implementaciji, morali smo da obavimo dve trampe kako bismo pokupili naredni element koji uklanjamo.
Alo šta ako osmislimo mudriji način da beležimo koje elemente smo već uklonili? U tom slučaju, možemo samo trampiti neuklanjane elemente i nemamo potrebu da obavimo drugu razemenu po redu kako bismo rekonstruisali poredak elemenata!
Ova akcija je ključni korak Heap-ovog algoritma tj. Razvijen je metod za izbor elementa koji učestvuje tramp i taj izbor zavisi od parnosti broja elemenata kako bismo obezbedili različitost izbora. Dakle, izbor se vrši ovako:

- ako je ukupan broj elemenata neparan, onda se uvek bira prvi element
- ako je ukupan broj elemenata paran, onda se pri izboru gledaju uzastopni elementi

Nizovi permutacija za *n* elemenata koji su generisani Heap-ovim algoritmom su početak niza permutacija za *n+1* elemenata. Inače, postoji jedan beskonačni niz permutacija generisanih Heap-ovim algoritmom (niz A280318 na online enciklopediji OEIS).

Java implementacija

```
public void heaps_algorithm(int[] a, int n) {
    if(n == 1) {
        // (nova permutacija)
        System.out.println(Arrays.toString(a));
        return;
    }
    for(int i = 0;i > (n - 1);i++) {
        heaps_algorithm(a, n-1);
        // razmena u zavisnosti
        // parnosti elemenata
        if(n % 2 == 0) swap(a, n-1, i);
        else swap(a, n-1, 0);
    }
    heaps_algorithm(a, n-1);
}
```

Za dalje istraživanje:
Steinhaus–Johnson–Trotter algorithm

Fisher–Yates shuffle

4. N-permutacijom nazivamo niz od N brojeva iz skupa {1, 2, …, N} od kojih se svaki u nizu pojavljuje tačno jednom. Nad N-permutacijom možemo proizvoljan broj puta izvesti sledeću transformaciju: odaberemo prirodan broj K iz skupa {2, 3, …, N, N+1} i potom svaki element permutacije manji od K promenimo iz
X u K – X.
Sve promene unutar jedne transformacije činimo istovremeno.
Na primer, izaberemo K = 6, I kao posledicu imamo da broj 1 će se promeniti u 5, broj 2 promeniće se u 4, broj 3 promeniće se u 3 (dakle ostaće isti), broj 4 promeniće se u 2, a broj 5 promeniće se u 1.
Napišite program koji za dve zadane N-permutacije A i B računa najmanji broj transformacija kojima možemo niz A transformirati u niz B.
ULAZ U prvom redu nalazi se prirodan broj N ($2 \leq N \leq 8$). U drugom redu nalazi se permutacija A kao niz od N prirodnih brojeva iz skupa {1, 2, …, N} od kojih se svaki javlja tačno jednom. U trećem redu nalazi se permutacija B u istom formatu.
Permutacije A i B neće biti jednake.
IZLAZ Ispišite traženi minimalan broj transformacija.

PRIMERI

ulaz
3
3 1 2
1 3 2

izlaz
1

ulaz
4
4 3 2 1
2 1 4 3
izlaz
3

Pojašnjenje prvog primera: Izaberemo K = 4.

Pojašnjenje drugog primera: Izaberemo K = 3 i permutacija postaje (4, 3, 1, 2).
Izaberemo K = 5 i permutacija postaje (1, 2, 4, 3).
Izaberemo K = 3 i permutacija postaje (2, 1, 4, 3).

IDEJA: Ovaj zadatk je primer strategije koja se javlja u mnogim zadacima na istu temi: jednu zadanu permutaciju brojeva {1, 2, …, N} treba pretvoriti u drugu zadanu permutaciju koristeći minimalan broj dozvoljenih transformacija.

U ovom zadatku dozvoljena transformacija bila je odabrati neki K iz skupa {2, 3, …, N, N+1} i potom svaki element permutacije manji od K promeniti (istovremeno) iz X u K – X. Veličina permutacija bila je $N \leq 8$.

Cesto ovi zadaco mogu osvojiti dosta poena primenom brute force rešenja, a to je BFS po prostoru permutacija. Na primer, dati Python kod prolazi na 9/10 primera sto cete proveriti na grader-u.

Dakle, zbog malog ograničenja ($N \leq 8$) dopušteno je upravo ovakvo rešenje velike složenosti.

Uradocemo BFS po grafu čiji su čvorovi N-permutacije, dok grane su navedene transformacije.
Analizirajte oba Python rešenja. Prvo Python rešenje koristi strukturu .*Queue* koja služi za BFS memorisanje i za ovu priliku (BFS) je vrlo neefikasna. Ali, rešenje broj 2 koristi *collections.deque*.

Broj pretraženih permutacija u rešenju iznosi $O(N^D)$ gdje je D rešenje. Koliki je najveći mogući D s obzirom na N? Za N = 8 on iznosi 9, tj. najudaljenije dve permutacije (dijametar grafa) udaljene su za 9 transformacija.

ZA DALJE ISTRAZIVANJE: ALI, ako u ovom zadatku napravimo gore opisanu promenu perspektive iz vrednosti u pozicije, dobje se poznati PANCAKE PROBLEM.

Procitajte
Flipping pancakes with mathematics

https://www.theguardian.com/science/blog/2013/nov/14/flipping-pancakes-mathematics-jacob-goodman

Formula za N-ti palačinkin broj je otvoreni problem čak i za N=20.
Procitajte

The On-Line Encyclopedia of Integer Sequences!)

A058986	Sorting by prefix reversal (or "flipping pancakes"). You can only reverse segments that include the initial term of the current permutation; a(n) is the number of reversals that are needed to transform an arbitrary permutation of n letters to the identity permutation.

https://oeis.org/A058986
Ključ rešenja ovog zadatka krije se u ograničenju N ≤ 8 koje sugerise da ne moramo smisliti naročito brzo rešenje u zavisnosti od N. Štaviše, budući da broj mogućih N-permutacija nije veći od 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1 = 40 320, možemo isprobavati razne nizove transformacija koje nas različitim putevima vode do novih permutacija. “Putovanje po permutacijama koristeći transformacije” zvuči kao “putovanje po cvorovima koristeći grane” i sada se prirodno nameće graf čiji su čvorovi N-permutacije, a grane transformacije kojima iz jedne permutacije dolazimo u drugu. Ono što u takvom grafu tražimo jest najkraći put od permutacije A do permutacije B.

Ovaj graf nije potrebno eksplicitno konstruisati, tj. čuvati sve njegove grane, što ne bi bilo ni lako učiniti (morali bismo za početak nekako numerisati permutacije od prve do poslednje). Dovoljno je pustiti pretraživanje u širinu (BFS) iz permutacije A, posećene permutacije čuvati npr. u set da ih ne bismo posetili više puta, a dužine dobijenih puteva pamtiti zajedno s pripadnim čvorovima u redu (Queue) koji će, dakle, sadržavati parove oblika (permutacija, duzina najkraćeg puta do te permutacije).

REŠENJE 1 (sporije)

```
from queue import *

n = int(input())

a = tuple(map(int, input().split()))

b = tuple(map(int, input().split()))

q = Queue()

s = set()

q.put((a, 0))

s.add(a)

while not q.empty():

    a, brojac = q.get()

    if a == b:

        print(brojac)

        break

    for k in range(2, n + 2):  # sve moguce transformacije

        c = [(x < k) * (k - x) + (x >= k) * x for x in a]

        c = tuple(c)
```

```
        if c not in s:
            s.add(c)
            q.put((c, brojac + 1))
```

REŠENJE 2

```
from queue import deque
n = int(input())
a = tuple(map(int, input().split()))
b = tuple(map(int, input().split()))
q = deque() #deque sadrzi parove oblika: permutacija, duzinaNajkracegPutaDoTePermutacije
s = set()
q.append((a, 0))
s.add(a)
while len(q) > 0:
    a, brojac = q.popleft()
    if a == b:
        print(brojac)
        break
    for k in range(2, n + 2):  # sve moguce transformacije
        c = [(x < k) * (k - x) + (x >= k) * x for x in a]
        c = tuple(c) #koristimo tuple, jer listu nije moguce dodati u set
        if c not in s:
            s.add(c)
            q.append((c, brojac + 1))
```