Vremensko ograničenje	Memorijsko ograničenje	ulaz	izlaz
0,1 s	64 MB	standardni ulaz	standardni izlaz

Napiši program koji izračunava vrednost aritmetičkog izraza u kojem se javljaju prirodni brojevi i između njih operatori +, - i *. Npr. 3+4*5-7*2.

Ulaz: Jedina linija standardnog ulaza sadrži opisani izraz.

Izlaz: Standardni izlaz treba da sadrži samo traženu vrednost učitanog izraza.

Primer

Ulaz

```
3+4*5-7*2
```

Izlaz

```

```

Python resenje

```
print(eval(input()))
```

2.

Prebroj izraze

Vremensko ograničenje	Memorijsko ograničenje	ulaz	izlaz
0,1 s	64 MB	standardni ulaz	standardni izlaz

Dat je string s **koji sadrži samo cifre (**0**, ...,** 9**) i prirodan broj** x**. Napisati program kojim se određuje broj izraza koji se mogu dobiti umetanjem operatora +, − i · u stringu** s **tako da je vrednost dobijenog izraza jednaka** x**. Pri tom, svaki operand u tom izrazu mora da bude ispravno zapisan prirodan broj (višecifreni brojevi ne smeju da počinju nulom).**

Ulaz

U prvoj liniji standardnog ulaza nalazi se string s**, druga linija sadrži prirodan broj** x**.**

Izlaz

Na standarnom izlazu u jednoj liniji prikazati broj traženih izraza.

Primer

Ulaz

1009

10

Izlaz

8

Objašnjenje: Postoje osam traženih izraza, to su izrazi: 1+0+0+9, 1+0−0+9, 1+0·0+9, 1−0+0+9, 1−0−0+9, 1−0·0+9, 10+0·9, 10−0·9

Rešenje

Jedan od najdirektnijih načina da se reši zadatak je da se generišu svi mogući opisani izrazi, da se izračuna vrednost svakoga i da se prebroje oni koji imaju traženu vrednost.

Generisanje svih izraza je slično postupku generisanja svih varijacija (koji smo prikazali u zadatku).

Postupno gradimo nisku karaktera koja sadrži izraz.

Generisanje počinjemo od izraza koji sadrži samo prvu cifru originalne niske.

U svakom koraku tekući izraz proširujemo tekućim karakterom originalne niske cifara i za svako takvo proširenje rekurzivno nastavljamo isti postupak sve dok se karakteri originalne niske ne iscrpe.

Proširenje tekućeg izraza cifrom možemo ostvariti na 4 načina:
1. samo dopisujući tu cifru ili
2. dopisujući tu cifru nakon svakog od 3 dopuštena operatora (na primer, izraz 32+4 možemo proširiti cifrom 5 tako da dobijemo izraze 32+45, 32+4+5, 32+4-5 i 32+4*5).
Prvi slučaj (dopisivanje cifre bez operatora) nije dopušten ako se tekući izraz završava jednocifrenim brojem 0.

```
def var(i, x, n, s, k):
    if(i == n):
        t = eval(x)
        if t == k:
            return 1
```

```
            else:
                return 0
    sum = 0
    if i != 0:
            sum += var(i+1, x+'+'+s[i], n, s, k)
            sum += var(i+1, x+'-'+s[i], n, s, k)
            sum += var(i+1, x+'*'+s[i], n, s, k)
    if i == 0 or s[i-1] != '0':
            sum += var(i+1, x+s[i], n, s, k)
    return sum


s = input()
k = int(input())
n = len(s)
print(var(0, "", n, s, k))
```

==========

```
#include <iostream>
#include <string>
#include <cctype>

using namespace std;

long long vrednost(const string& s) {
  long long rezultat = 0;
  int znakTekucegSabirka = 1;
  long long tekuciSabirak = 1;
  long long tekuciBroj = 0;
  for (int i = 0; i <= s.length(); i++)
    if (i < s.length() && isdigit(s[i]))
      // procitali smo cifru
      // dodajemo je kao poslednju cifru tekuceg broja
      tekuciBroj = 10 * tekuciBroj + s[i] - '0';
    else {
      // dosli smo do nekog operatora ili kraja broja

      // tekuci broj je faktor tekuceg sabirka
      tekuciSabirak *= tekuciBroj;
      // zavrsili smo sa obradom tekuceg broja i priremamo se za citanje narednog
      tekuciBroj = 0;

      // ako smo stigli do kraja ili procitali aditivni operator
      // zavrsili smo obradu tekuceg sabirka
      if (i == s.length() || s[i] == '+' || s[i] == '-') {
```

```
        // rezultat uvecavamo ili umanjujemo za tekuci sabirak u
        // zavisnosti od ranije odredjenog znaka
        rezultat += znakTekucegSabirka * tekuciSabirak;
        if (i < s.length()) {
          // priremamo se za obradu narednog sabirka
          tekuciSabirak = 1;
          // njegov znak postavljamo u zavisnosti od operatora na koji
          // smo naisli
          znakTekucegSabirka = s[i] == '+' ? 1 : -1;
        }
      }
    }
  return rezultat;
}

// odredjuje broj nacina da se dobije vrednost x prosirivanjem izraza
// tekuciIzraz koriscenjem karaktera iz s pocevsi od pozicije poz
int brojNacina(const string& cifre, int poz, int x, string tekuciIzraz) {
  if (poz == cifre.length())
    // obradili smo sve karaktere niske s, pa se tekuci izraz ne moze
    // dalje prosirivati
    // broj nacina da se dobije x prosirivanjem tekuceg izraza je 1 ako
    // je vrednost tog izraza jednaka broju x, tj. 0 u suprotnom
    return vrednost(tekuciIzraz) == x ? 1 : 0;

  // broj nacina na koji mozemo prosiriti izraz tako da mu je vrednost x
  int ukupanBrojNacina = 0;

  // tekuci izraz prosirujemo cifrom s[poz]

  // ako se tekuci izraz ne zavrsava jednocifrenim brojem 0
  if (tekuciIzraz[tekuciIzraz.length() - 1] != '0' ||
      (tekuciIzraz.length() > 1 && isdigit(tekuciIzraz[tekuciIzraz.length() - 2])))
    // cifru samo dopisujemo na tekuci izraz
    ukupanBrojNacina += brojNacina(cifre, poz+1, x, tekuciIzraz + cifre[poz]);

  // cifru dopisujemo iza svakog od dopustenih operatora
  ukupanBrojNacina += brojNacina(cifre, poz+1, x, tekuciIzraz + "+" +
cifre[poz]);
  ukupanBrojNacina += brojNacina(cifre, poz+1, x, tekuciIzraz + "-" + cifre[poz]);
  ukupanBrojNacina += brojNacina(cifre, poz+1, x, tekuciIzraz + "*" +
cifre[poz]);
  return ukupanBrojNacina;
}

// broj nacina da se umetanjem operatora +, -, * u cifre date u nizu s
// dobije izraz cija je vrednost x
int brojNacina(const string& cifre, int x) {
  // tekuci izraz inicijalizujemo na prvu cifru
  return brojNacina(cifre, 1, x, string(1, cifre[0]));
}

int main() {
  string cifre;
```

```
  cin >> cifre;
  int ciljnaVrednost;
  cin >> ciljnaVrednost;
  cout << brojNacina(cifre, ciljnaVrednost) << endl;
  return 0;
}
```

Izračunavanje vrednosti svih izraza bez njihovog generisanja

Prethodno rešenje se može unaprediti. Jedna od mana je to što se u rešenju manipuliše sa niskama koje čuvaju tekući izraz, što može biti neefikasno. Druga, još upadljivija mana je to što puno izgrađenih izraza deli zajednički prefiks čija se vrednost nepotrebno izračunava mnogo puta. Osnova unapređenja rešenja je to da se kroz rekurziju umesto niske koja predstavlja tekući izraz šalje vrednost tekućeg izraz.

Prvi korak je da se odabere prvi broj u izrazu koji gradimo. Ako je prva cifra date niske nula, onda je jedini način da prvi broj bude 0. U suprotnom prvi broj može biti bilo koji neprazan prefiks naše niske i tada se ukupan broj načina može dobiti sabiranjem broja načina za svaki izbor prvog broja (na primer, ako je niska 123 sabiramo koliko ima izraza koji počinju sa 1, koliko sa 12, a koliko sa 123).

Nakon određivanja prvog broja krećemo da postavljamo operatore. Ako je potrebno da postavimo operator na poziciju p tada biramo drugi operand i to u delu niske koji počinje na poziciji p. Njega biramo na isti način kao i prvi broj. Ako je cifra na poziciji p nula, tada je jedini način da se taj operand izabere baš ta nula, a u suprotnom je moguće odabrati bilo koji neprazan prefiks dela niske koji počinje na poziciji p. Kada je odabran drugi operand, vrednost prethodnog izraza treba proširiti njime i to tako da se između nalazi bilo koji od tri dopuštena operatora. U slučaju operatora + i - vrednost proširenog izraza se direktno ažurira. U slučaju operatora * vrednost novog izraza se dobija tako što se od prethodne vrednosti izraza oduzme vrednost poslednjeg sabirka a zatim doda vrednost tog sabirka pomnožena sa vrednošću drugog operanda. Zbog toga je uz vrednost tekućeg izraza kroz rekurzivne pozive potrebno prosleđivati i vrednost poslednjeg uračunatog sabirka. Ovo u potpunosti odgovara postpuku spekulativnog izračunavanja vrednosti podizraza koji smo opisali u zadatku Nezagrađen izraz.

```
#include <iostream>
#include <string>

using namespace std;

// Neka je poznat izraz e + c, takav da mu je vrednost jednaka broju
// vrednost, a da je vrednost njegovog poslednjeg sabirka
// jednaka broju poslednjiSabirak. Funkcija izracunava broj nacina da
// se taj izraz prosiri ciframa niske s, pocevsi od pozicije poz, tako
// da prosireni izraz ima vrednost jednaku broju ciljnaVrednost.
int brojNacina(const string& cifre, int poz,
               long long vrednost, long long poslednjiSabirak,
```

```
            int ciljnaVrednost) {
  // ako nema vise karaktera izraz se ne moze prosiriti i proveravamo
  // da li je njegova vrednost jednaka ciljnoj
  if (poz == cifre.length())
    return vrednost == ciljnaVrednost ? 1 : 0;

  // ukupan broj nacina
  int ukupanBrojNacina = 0;
  // odredjujemo broj kojim prosirujemo izraz (tako sto izmedju izraza
  // i tog broja umecemo neki od dopustenih operatora)
  long long broj = 0;
  for (int i = poz; i < cifre.length(); i++) {
    // ako je cifre[poz] = 0, tada drugi operand moze biti samo 0, sto
je
    // slucaj vec obradjen u prvoj iteraciji petlje, pa se petlja moze
    // prekinuti
    if (cifre[poz] == '0' && i > poz)
      break;

    // tekuci broj prosirujemo jos jednom cifrom
    broj = broj * 10 + cifre[i] - '0';

    // tako odredjen drugi operand kombinujemo sa izrazom pomocu
    // svakog od tri dopustena operatora, rekurzivno izracunavamo
    // odgovarajuca prosirenja tako dobijenih izraza i sve dobijene
    // ukupanBrojNacinae sabiramo
    ukupanBrojNacina += brojNacina(cifre, i+1, vrednost + broj, broj,
ciljnaVrednost);
    ukupanBrojNacina += brojNacina(cifre, i+1, vrednost - broj, -broj,
ciljnaVrednost);
    ukupanBrojNacina += brojNacina(cifre, i+1,
                           vrednost - poslednjiSabirak + poslednjiSabirak
* broj, poslednjiSabirak * broj,
                           ciljnaVrednost);
  }
  // vracamo ukupan broj nacina
  return ukupanBrojNacina;
}

// broj nacina da se umetanjem operatora +, -, * izmedju cifara dobije
// izraz cija je vrednost jednaka broju ciljnaVrednost
int brojNacina(const string& cifre, int ciljnaVrednost) {
  // ukupan broj nacina da se napravi izraz sa ciljnom vrednoscu
  int ukupanBrojNacina = 0;
  // odredjujemo sve moguce prve clanove tog izraza
  long long broj = 0;
  for (int i = 0; i < cifre.length(); i++) {
    // ako izraz pocinje nulom, onda je jedini moguci prvi clan bas ta
    // nula - ako je ona obradjena u prethodnoj iteraciji, petlja se
    // moze zaustaviti
    if (cifre[0] == '0' && i > 0)
      break;

    // prosirujemo tekuci broj tekucom cifrom
    broj = broj * 10 + cifre[i] - '0';
    // izracunavamo broj nacina da se dobije ciljna vrednost
    // prosirivanjem izraza koji se sastoji samo od tog pocetnog broja
```

```
    ukupanBrojNacina += brojNacina(cifre, i + 1, broj, broj,
ciljnaVrednost);
  }
  return ukupanBrojNacina;
}

int main() {
   string cifre;
   cin >> cifre;
   int ciljnaVrednost;
   cin >> ciljnaVrednost;
   cout << brojNacina(cifre, ciljnaVrednost) << endl;
   return 0;
}
```

RESENJE C#

```
using System;

class Program
{

    static long vrednost(string s) {
        long rezultat = 0;
        int znakTekucegSabirka = 1;
        long tekuciSabirak = 1;
        long tekuciBroj = 0;
        for (int i = 0; i <= s.Length; i++)
            if (i < s.Length && Char.IsDigit(s[i]))
                // procitali smo cifru dodajemo je kao poslednju
cifru
                // tekuceg broja
                tekuciBroj = 10 * tekuciBroj + s[i] - '0';
            else {
                // dosli smo do nekog operatora ili kraja broja

                // tekuci broj je faktor tekuceg sabirka
                tekuciSabirak *= tekuciBroj;
                // zavrsili smo sa obradom tekuceg broja i priremamo
                // se za citanje narednog
                tekuciBroj = 0;

                // ako smo stigli do kraja ili procitali aditivni
                // operator zavrsili smo obradu tekuceg sabirka
                if (i == s.Length || s[i] == '+' || s[i] == '-') {
                    // rezultat uvecavamo ili umanjujemo za tekuci
                    // sabirak u zavisnosti od ranije odredjenog
znaka
                    rezultat += znakTekucegSabirka * tekuciSabirak;
                    if (i < s.Length) {
                        // priremamo se za obradu narednog sabirka
                        tekuciSabirak = 1;
                        // njegov znak postavljamo u zavisnosti od
                        // operatora na koji smo naisli
                        znakTekucegSabirka = s[i] == '+' ? 1 : -1;
                    }
                }
```

```
            }
        return rezultat;
    }

    // odredjuje broj nacina da se dobije vrednost x prosirivanjem
    // izraza tekuciIzraz koriscenjem karaktera iz s pocevsi od
    // pozicije poz
    static int brojNacina(string cifre, int poz, int x, string
tekuciIzraz) {
        if (poz == cifre.Length)
            // obradili smo sve karaktere niske s, pa se tekuci
izraz
            // ne moze dalje prosirivati broj nacina da se dobije x
            // prosirivanjem tekuceg izraza je 1 ako je vrednost tog
            // izraza jednaka broju x, tj. 0 u suprotnom
            return vrednost(tekuciIzraz) == x ? 1 : 0;

        // broj nacina na koji mozemo prosiriti izraz tako da mu je
        // vrednost x
        int ukupanBrojNacina = 0;

        // tekuci izraz prosirujemo cifrom s[poz]

        // ako se tekuci izraz ne zavrsava jednocifrenim brojem 0
        if (tekuciIzraz[tekuciIzraz.Length - 1] != '0' ||
            (tekuciIzraz.Length > 1 &&
Char.IsDigit(tekuciIzraz[tekuciIzraz.Length - 2])))
            // cifru samo dopisujemo na tekuci izraz
            ukupanBrojNacina += brojNacina(cifre, poz+1, x,
tekuciIzraz + cifre[poz]);

        // cifru dopisujemo iza svakog od dopustenih operatora
        ukupanBrojNacina += brojNacina(cifre, poz+1, x, tekuciIzraz
+ "+" + cifre[poz]);
        ukupanBrojNacina += brojNacina(cifre, poz+1, x, tekuciIzraz
+ "-" + cifre[poz]);
        ukupanBrojNacina += brojNacina(cifre, poz+1, x, tekuciIzraz
+ "*" + cifre[poz]);
        return ukupanBrojNacina;
    }

    // broj nacina da se umetanjem operatora +, -, * u cifre date u
    // nizu s dobije izraz cija je vrednost x
    static int brojNacina(string cifre, int x) {
        // tekuci izraz inicijalizujemo na prvu cifru
        return brojNacina(cifre, 1, x, cifre[0].ToString());
    }


    static void Main(string[] args)
    {
        string cifre = Console.ReadLine();
        int ciljnaVrednost = int.Parse(Console.ReadLine());
        Console.WriteLine(brojNacina(cifre, ciljnaVrednost));
    }
```

}

3. Broj podnizova datog zbira

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

Napiši program koji određuje koliko podnizova (ne obavezno uzastopnih elemenata) datog niza pozitivnih brojeva ima zbir jednak datom broju.

Ulaz

Sa standardnog ulaza se učitava broj $1 \leq n \leq 30$**, a zatim u narednom redu** n **pozitivnih realnih brojeva razdvojenih razmacima.**

Izlaz

Na standardni izlaz ispisati traženi broj podnizova (dva realna broja se mogu smatrati jednakima ako se razlikuju za manje od 10^{-5}**).**

Primer

Ulaz
```
4
3.2
5.7
9.4
6.9
12.6
```
Izlaz
```
2
```
Obrazloženje: Važi da je $3,2+9,4=5,7+6,9=12,6$.
Izaberi programski jezik

Jedna mogućnost je da se zadatak reši grubom silom, tj. da se nabroje svi podnizovi i da se za svaki od njih proveri da li mu je zbir jednak traženom. Nabrajanje podnizova se može postići na bilo koji od načina prikazanih u zadatku sa proslog casa Svi podskupovi. Jedna mogućnost implementacije podrazumeva da čuvamo elemente podniza, trenutni broj elemenata podniza i parametar koji određuje koji deo skupa je još potrebno obraditi (to može biti dužina prefiksa niza koji još nije obrađen). Kada se taj prefiks isprazni, proveravamo tekući podniz. U suprotnom rekurzivno razmatramo mogućnost da poslednji element prefiksa nije ili jeste uključen u podniz, uklanjajući u oba slučaja taj poslednji element iz prefiksa (tako što se smanjuje njegova dužina).

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

using namespace std;

const double EPS = 0.00001;
```

```
// racuna se broj podnizova koji elementima niza na pozicijama [k, n)
// nastavljaju dati podniz duzine k i koji imaju dati zbir
int brojPodnizovaDatogZbira(const vector<double>& niz, int n,
                            double ciljniZbir,
                            vector<double>& podniz, int k) {
    // u nizu nema preostalih elemenata, pa je trenutni podniz jedini kandidat
    if (n == 0) {
        // racunamo zbir trenutnog podniza
        double zbirPodniza = 0.0;
        for (int i = 0; i < k; i++)
            zbirPodniza += podniz[i];
        // proveravamo da li je jednak ciljnom zbiru
        if (abs(zbirPodniza - ciljniZbir) < EPS)
            return 1;
        else
            return 0;
    } else {
        // broj podnizova bez ukljucenog poslednjeg elementa niza
        int broj = 0;
        broj += brojPodnizovaDatogZbira(niz, n-1, ciljniZbir, podniz, k);
        // broj podnizova sa ukljucenim poslednjim elementom niza
        podniz[k] = niz[n-1];
        broj += brojPodnizovaDatogZbira(niz, n-1, ciljniZbir, podniz, k+1);
        return broj;
    }
}

// funkcija racuna koliko podnizova datog niza ima zbir jednak ciljnom
int brojPodnizovaDatogZbira(vector<double>& niz, double ciljniZbir) {
    // broj elemenata niza
    int n = niz.size();
    vector<double> podniz(n);
    // rekurzivnom funkcijom racunamo trazeni broj podnizova
    return brojPodnizovaDatogZbira(niz, n, ciljniZbir, podniz, 0);
}

int main() {
    // ucitavamo niz
    int n;
    cin >> n;
    vector<double> niz(n);
    for (int i = 0; i < n; i++)
        cin >> niz[i];
    // ciljni zbir podniza
    double ciljniZbir;
```

```
    cin >> ciljniZbir;

    // izracunavamo i ispisujemo trazeni broj podnizova
    cout << brojPodnizovaDatogZbira(niz, ciljniZbir) << endl;

    return 0;
}
```

Druga mogućnost implementacije pretrage grubom silom je da kao parametar rekurzivne funkcije prosleđujemo trenutni ciljni zbir tj. razliku između traženog zbira i zbira elemenata trenutno uključenih u podniz. Sam podniz nije neophodno održavati. Ako je ciljni zbir jednak nuli, to znači da je zbir trenutnog podniza jednak traženom i da smo našli jedan zadovoljavajući podniz. U suprotnom, ako u skupu nema preostalih elemenata, tada znamo da nije moguće napraviti podniz traženog zbira. U suprotnom uklanjamo trenutni element iz niza i razmatramo mogućnost da se on uključi i mogućnost da se ne uključi u podniz. U prvom slučaju umanjujemo ciljni zbir za vrednost tog elementa, a u drugom ciljni zbir ostaje nepromenjen.

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

using namespace std;

const double EPS = 0.00001;

// funkcija odredjuje broj podnizova niza odredjenog elementima na
// pozicijama [k, n) takvih da je zbir elemenata podniza jednak
ciljnom zbiru
int brojPodnizovaDatogZbira(const vector<double>& niz, double
ciljniZbir, int k) {
    // jedino prazan niz ima zbir nula
    if (abs(ciljniZbir) < EPS)
        return 1;

    // jedini podniz praznog niza je prazan, a ciljni zbir je pozitivan
    if (k == niz.size())
        return 0;

    // posebno brojimo podnizove koji ukljucuju niz[k] i one koji ne
ukljucuju
    // niz[k]
    return brojPodnizovaDatogZbira(niz, ciljniZbir - niz[k], k+1) +
           brojPodnizovaDatogZbira(niz, ciljniZbir, k+1);
}

int brojPodnizovaDatogZbira(const vector<double>& niz, double
ciljniZbir) {
    // brojimo podnizove niza odredjenog elementima na pozicijama [0, n)
    return  brojPodnizovaDatogZbira(niz, ciljniZbir, 0);
}
```

```
int main() {
    // ucitavamo niz
    int n;
    cin >> n;
    vector<double> niz(n);
    for (int i = 0; i < n; i++)
        cin >> niz[i];
    // ucitavamo ciljni zbir
    double ciljniZbir;
    cin >> ciljniZbir;

    // izracunavamo i ispisujemo trazeni broj podnizova
    cout << brojPodnizovaDatogZbira(niz, ciljniZbir) << endl;

    return 0;
}
```

Efikasnija rešenja od rešenja grubom silom se mogu dobiti primenom različitih odsecanja. Ključna stvar je da odredimo interval u kome mogu ležati zbirovi svih podnizova preostalih elemenata niza. Pošto su svi elementi pozitivni, najmanja moguća vrednost zbira podniza je nula (u slučaju praznog niza), dok je najveća moguća vrednost zbira podniza jednaka zbiru svih elemenata niza. Dakle, ako je ciljni zbir strogo manji od nule ili strogo veći od zbira svih elemenata preostalog dela niza, tada ne postoji ni jedan podniz čiji je zbir jednak ciljnom. Umesto da zbir svih elemenata niza računamo iznova u svakom rekurzivnom pozivu, možemo primetiti da se u svakom narednom rekurzivnom pozivu niz samo može smanjiti za jedan element, pa se zbir može računati inkrementalno, umanjivanjem tokom rekurzije zbira polaznog niza za elemente uklonjene iz niza.

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

using namespace std;

const double EPS = 0.00001;

// racuna se broj podnizova elemenata niza na pozicijama [k, n) koji
// imaju dati zbir, pri cemu se zna da je zbir tih elemenata jednak
// zbirPreostalih
int brojPodnizovaDatogZbira(const vector<double>& niz, double
ciljniZbir,
                            double zbirPreostalih, int k) {
    // ciljni zbir 0 se dobija samo ako se ne uzme ni jedan element
    if (abs(ciljniZbir) < EPS)
        return 1;

    // jedini podniz praznog niza je prazan, a ciljni zbir je pozitivan
    if (k == niz.size())
        return 0;
```

```
    // posto su svi brojevi pozitivni, nije moguce dobiti negativan
ciljni zbir
    if (ciljniZbir + EPS < 0)
        return 0;

    // cak ni uzimanje svih elemenata ne moze dovesti do ciljnog zbira,
    // pa nema podnizova koji bi dali ciljni zbir
    if (zbirPreostalih + EPS < ciljniZbir)
        return 0;

    // broj podnizova u kojima ucestvuje element a[k]
    return brojPodnizovaDatogZbira(niz, ciljniZbir - niz[k],
                                   zbirPreostalih - niz[k], k+1) +
        // broj podnizova u kojima ne ucestvuje element a[k]
        brojPodnizovaDatogZbira(niz, ciljniZbir,
                                zbirPreostalih - niz[k], k+1);
}

// funkcija racuna koliko podnizova datog niza ima zbir jednak
ciljnom
int brojPodnizovaDatogZbira(vector<double>& niz, double ciljniZbir) {
    // broj elemenata niza
    int n = niz.size();
    // izracunavamo zbir elemenata niza
    double zbirNiza = 0;
    for (int i = 0; i < n; i++)
        zbirNiza += niz[i];
    // rekurzivnom funkcijom racunamo trazeni broj podnizova
    return  brojPodnizovaDatogZbira(niz, ciljniZbir, zbirNiza, 0);
}


int main() {
    // ucitavamo niz
    int n;
    cin >> n;
    vector<double> niz(n);
    for (int i = 0; i < n; i++)
        cin >> niz[i];
    // ucitavamo ciljni zbir
    double ciljniZbir;
    cin >> ciljniZbir;

    // izracunavamo i ispisujemo trazeni broj podnizova
    cout << brojPodnizovaDatogZbira(niz, ciljniZbir) << endl;

    return 0;
}
```

Još jedno moguće odsecanje se može izvršiti kada se ustanovi da je najmanji od preostalih brojeva u nizu veći od ciljnog zbira. Ako je taj ciljni zbir pozitivan, tada

nije moguće dostići ga (jer prazan podniz ima zbir nula, a bilo koji neprazan podniz ima zbir veći ili jednak od tog minimalnog elementa). Minimalni element preostalog dela niza je jednostavno odrediti ako se niz sortira (što možemo uraditi pre početka pretrage).

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

using namespace std;

const double EPS = 0.00001;

// racuna se broj podnizova elemenata niza na pozicijama [k, n) koji
// imaju dati zbir, pri cemu se zna da je zbir tih elemenata jednak
// zbirPreostalih
int brojPodnizovaDatogZbira(const vector<double>& niz, double
ciljniZbir,
                            double zbirPreostalih, int k) {
    // ciljni zbir 0 se dobija samo ako se ne uzme ni jedan element
    if (abs(ciljniZbir) < EPS)
        return 1;

    // jedini podniz praznog niza je prazan, a ciljni zbir je pozitivan
    if (k == niz.size())
        return 0;

    // cak ni uzimanje svih elemenata ne moze dovesti do ciljnog zbira,
    // pa nema podnizova koji bi dali ciljni zbir
    if (zbirPreostalih + EPS < ciljniZbir)
        return 0;

    // vec uzimanje najmanjeg elementa prevazilazi ciljni zbir, pa
    // nema podnizova koji bi dali ciljni zbir
    if (niz[k] > ciljniZbir + EPS)
        return 0;

            // broj podnizova u kojima ucestvuje element a[k]
    return brojPodnizovaDatogZbira(niz, ciljniZbir - niz[k],
                                   zbirPreostalih - niz[k], k+1) +
            // broj podnizova u kojima ne ucestvuje element a[k]
            brojPodnizovaDatogZbira(niz, ciljniZbir,
                                    zbirPreostalih - niz[k], k+1);
}

// funkcija racuna koliko podnizova datog niza ima zbir jednak
ciljnom
int brojPodnizovaDatogZbira(vector<double>& niz, double ciljniZbir) {
    // broj elemenata niza
    int n = niz.size();
```

```
    // sortiramo elemente niza neopadajuce
    sort(begin(niz), end(niz));
    // izracunavamo zbir elemenata niza
    double zbirNiza = 0;
    for (int i = 0; i < n; i++)
        zbirNiza += niz[i];
    // rekurzivnom funkcijom racunamo trazeni broj podnizova
    return  brojPodnizovaDatogZbira(niz, ciljniZbir, zbirNiza, 0);
}


int  main() {
    // ucitavamo niz
    int n;
    cin >> n;
    vector<double> niz(n);
    for (int i = 0; i < n; i++)
        cin >> niz[i];
    // ciljni zbir podniza
    double ciljniZbir;
    cin >> ciljniZbir;

    // izracunavamo i ispisujemo trazeni broj podnizova
    cout << brojPodnizovaDatogZbira(niz, ciljniZbir) << endl;

    return 0;
}
```

4. Broj kombinacija sa ponavljanjem

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

Napiši program koji određuje na koliko se načina može izvući k **loptica iz bubnja koji sadrži** n **različitih loptica, ako se svaka izvučena loptica vraća u bubanj pre novog izvlačenja.**

Ulaz
Sa standardnog ulaza se učitava broj k **(**1≤k≤n**) i** n **(**1≤n≤30**), svaki iz posebnog reda.**
Izlaz
Na standardni izlaz ispisati traženi broj kombinacija sa ponavljanjem.
Primer
Ulaz
2
3
Izlaz
6
Objašnjenje
To su kombinacije (1,1)**,** (1,2)**,** (1,3)**,** (2,2)**,** (2,3) **i** (3,3)

Ako je potrebno izabrati 0 **elemenata iz skupa od** n **elemenata to je moguće uraditi samo na jedan način. Ako je potrebno izabrati** k>0 **elemenata iz praznog skupa, to nije moguće učiniti. U suprotnom, sve kombinacije možemo podeliti na one koje počinju najmanjim elementom skupa od** n **elemenata i na one koji ne počinju njime. Možemo dakle, uzeti najmanji element skupa i zatim gledati sve moguće načine da se odabere** k−1 **element iz istog** n**-točlanog skupa ili možemo svih** k **elemenata izabrati iz skupa iz kog je izbačen taj najmanji element. Ovim dobijamo veoma jednostavnu rekurentu vezu na osnovu koje možemo implementirati rekurzivnu funkciju (koja će, doduše, biti neefikasna).**
f(0,n)=1 f(k,0)=0,za k>0 f(k,n)=f(k−1,n)+f(k,n−1),za k,n>0

```
#include <iostream>

using namespace std;

int brojKombinacijaSaPonavljanjem(int k, int n) {
    if (k == 0) return 1;
    if (n == 0) return 0;
    return brojKombinacijaSaPonavljanjem(k-1, n) +
           brojKombinacijaSaPonavljanjem(k, n-1);
}

int main() {
    int K, N;
    cin >> K >> N;
    cout << brojKombinacijaSaPonavljanjem(K, N) << endl;
    return 0;
}
```

Neefikasnost prethodne funkcije potiče od toga što se identični rekurzivni pozivi ponavljaju više puta. To je moguće popraviti tehnikom dinamičkog programiranja. Broj kombinacija za razne vrednosti (n,k) **se može rasporediti u pravougaonik.**

```
                              (k,n)
0  1   1   1   1   1          (0,0) (0,1) (0,2) (0,3) (0,4) (0,5)
0  1   2   3   4   5          (1,0) (1,1) (1,2) (1,3) (1,4) (1,5)
0  1   3   6  10  15          (2,0) (2,1) (2,2) (2,3) (2,4) (2,5)
0  1   4  10  20  35          (3,0) (3,1) (3,2) (3,3) (3,4) (3,5)
0  1   5  15  35  70          (4,0) (4,1) (4,2) (4,3) (4,4) (4,5)
0  1   6  21  56 126          (5,0) (5,1) (5,2) (5,3) (5,4) (5,5)
```

Prethodna rekurentna veza nam govori da se u prvoj koloni nalaze nule, u prvoj vrsti jedinice, a da je svaki element u ostatku matrice jednak zbiru elemenata neposredno iznad i levo od njega. Matricu možemo popuniti vrstu po vrstu, no, možemo napraviti i narednu memorijsku optimizaciju.

Elementi svake vrste zavise samo od elemenata te i prethodne vrste, pa nije neophodno čuvati celu matricu, već je dovoljno čuvati samo jedan niz, koji će tokom ažuriranja čuvati neke elemente prethodne i neke elemente tekuće vrste. Vrsta za sledeće k se može dobiti ažuriranjem vrste za prethodno k. Pošto svaki element u pravougaoniku zavisi od vrednosti iznad i levo od sebe, vrstu možemo ažurirati sleva nadesno. Invarijanta je da se u trenutku ažuriranja pozicije n, na pozicijama strogo manjim od n nalaze vrednosti iz tekuće vrste k, a na pozicijama od n nadalje se nalaze vrednosti iz prethodne vrste $k-1$. Element na poziciji n koji sadrži vrednost sa pozicije $(k-1,n)$ pravougaonika uvećavamo za vrednost levo od njega koji sadrži vrednost $(k,n-1)$ i tako dobijamo vrednost elementa na poziciji (k,n). Uvećavanjem vrednosti n za 1 se održava invarijanta.

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

using namespace std;

long long brojKombinacijaSaPonavljanjem(int K, int N) {
    vector<long long> dp(N+1, 1);
    dp[0] = 0;
    for (int k = 1; k <= K; k++)
        for (int n = 1; n <= N; n++)
            dp[n] += dp[n-1];
    return dp[N];
}

int main() {
    int K, N;
    cin >> K >> N;
    cout << brojKombinacijaSaPonavljanjem(K, N) << endl;
    return 0;
}
```

Postoji i veoma zgodan algoritam da se kombinacije sa ponavljanjem svedu na obične kombinacije (čiji smo broj računali u zadatku iz prosle godine, Broj kombinacija). Zamislimo da smo poređali sve elemente od 1 do n i da pravimo program za robota koji će odabrati kombinaciju tako što kreće od prvog broja i u svakom trenutku ili može da uzme taj broj (operacija $+$) ili da se pomeri na sledeći broj (operacija $\rightarrow$), sve dok ne stigne do poslednjeg broja, pri čemu treba ukupno da uzme k brojeva. Na primer, ako je niz brojeva $1, 2, 3, 4$ i bira se 4 broja, onda program $\rightarrow,+,+,\rightarrow,\rightarrow,+,+$ označava kombinaciju $2,2,4,4$. Pitanje, dakle, svodimo na to kako rasporediti $n-1$ strelica i k pluseva, što odgovara pitanju kako rasporediti k pluseva na $n+k-1$ pozicija, tako da je je ukupan broj kombinacija sa ponavljanjem jednak $(n+k-1k)$.

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

using namespace std;

long long brojKombinacija(int K, int N) {
```

```
    // tekuća vrsta
    vector<long long> dp(K+1);
    // na početku svake vrste nalazi se 1
    dp[0] = 1;
    // trougao popunjavamo kolonu po kolonu
    for (int n = 1; n <= N; n++) {
        // vrstu ažruriramo zdesna nalevo
        // na kraju svake vrste nalazi se 1
        if (n <= K) dp[n] = 1;
        // ažuriramo unutrašnje elemente
        for (int k = min(n-1, K); k > 0; k--)
            dp[k] += dp[k-1];
    }
    // vraćamo traženi rezultat
    return dp[K];
}

long long brojKombinacijaSaPonavljanjem(int K, int N) {
    return brojKombinacija(K, N + K - 1);
}

int main() {
    int K, N;
    cin >> K >> N;
    cout << brojKombinacijaSaPonavljanjem(K, N) << endl;
    return 0;
}
```

5. **Broj particija**

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

Particija pozitivnog prirodnog broja n **je rastavljanje broja** n **na zbir nekoliko pozitivnih prirodnih brojeva pri čemu je redosled sabiraka nebitan (stoga možemo pretpostaviti da je taj redosled ili uvek nerastući ili uvek neopadajući). Na primer, ako je redosled nerastući, particije broja** 4 **su** $1+1+1+1$, $2+1+1$, $2+2$, $3+1$, 4. **Napisati program koji određuje broj particija za dati prirodan broj** n.

Ulaz

Prva i jedina linija standardnog ulaza sadrži prirodan broj n **(**$n \leq 100$**).**

Izlaz

Na standardnom izlazu prikazati u prvoj liniji broj particija prirodnog broja n.

Primer 1

Ulaz

6

Izlaz

11

Objašnjenje

Ako su particije sa neopadajuće sortiranim sabircima, to su particije:

1+1+1+1+1+1

1+1+1+1+2

1+1+1+3

1+1+2+2

1+1+4

1+2+3

1+5

2+2+2

2+4

3+3

6

Primer 2

Ulaz

100

Izlaz

190569292

Rekurzivne procedure koje nabrajaju sve particije, su opisane u zadatku Sve particije (radjen pre dva casa) se mogu prilagoditi tako da izračunaju broj particija.

Svaka particija ima svoj prvi sabirak. Svakoj particiji broja n **kojoj je prvi sabirak** s **(pri čemu je** $1 \leq s \leq n$**) jednoznačno odgovara neka particija broja** $n-s$**. Nametnućemo uslov da su sabirci u svakoj particiji sortirani nerastuće. Zato, ako je prvi sabirak** s**, svi sabirci iza njega moraju da budu manji ili jednaki od** s**. Zato nam nije dovoljno samo da umemo da prebrojimo sve particije broja** $n-s$**, već je potrebno da ojačamo induktivnu hipotezu. Označimo sa** $p_{n,smax}$ **broj particija broja** n **u kojima su svi sabirci manji ili jednaki od** $smax$**.**

- **Bazu indukcije čini slučaj** $n=0$**, jer broj nula ima samo jednu particiju koja ne sadrži sabirke. Dakle, važi da je** $p_{0,smax}=1$**. Ako je** n **veće od nula i** $smax=0$**, tada ne postoji ni jedna particija, jer od sabiraka koji su svi jednaki nuli (jer svi moraju da budu manji ili jednaki** $smax$**) ne**

možemo nikako napraviti neki pozitivan broj. Dakle, za $n>0$ **važi da je** $p_{n,0}=0$.

- **Induktivni korak možemo ostvariti na više načina. Najjednostavniji je sledeći. Prilikom izračunavanja** $p_{n,smax}$ **možemo razmatrati dva slučaja: da se u zbiru ne javlja sabirak** $smax$ **ili da se u zbiru javlja sabirak** $smax$**. Ako se u zbiru ne javlja sabirak** $smax$**, tada je najveći sabirak** $smax-1$ **i broj takvih particija je** $p_{n,smax-1}$**. Drugi slučaj je moguć samo kada je** $n \geq smax$ **i broj takvih particija je** $p_{n-smax,smax}$**. Na osnovu ovoga, dobijamo narednu rekurzivnu funkciju.**

```
#include <iostream>
using namespace std;

// particije broja n u kojima je najveci sabirak jednak smax
int brojParticija(int n, int smax) {
  if (n == 0) return 1;
  if (smax == 0) return 0;
  int broj = brojParticija(n, smax - 1);
  if (n >= smax)
    broj += brojParticija(n - smax, smax);
  return broj;
}

int brojParticija(int n) {
  // particije broja n u kojima je najveći sabirak jednak n
  return brojParticija(n, n);
}

int main() {
  int n;
  cin >> n;
  cout << brojParticija(n) << endl;
}
```

Rekurzivnu funkciju za izračunavanje broja particija možemo dobiti i tako što na tekuće mesto u particiji postavljamo sve moguće kandidate za proširivanje tekuće particije (to su svi brojevi s **od** 1 **do manjeg od brojeva** $smax$ **i** n**) i nastavljamo generisanje particija broja** $n-s$ **kod kojih je najveći sabirak jednak** s**.**

```
#include <iostream>
#include <algorithm>
using namespace std;


// particije broja n u kojima je najveći sabirak jednak smax
int brojParticija(int n, int smax) {
    if (n == 0) return 1;
    if (smax == 0) return 0;
    int broj = 0;
    for (int s = min(smax, n); s >= 1; s--)
      broj += brojParticija(n - s, s);
    return broj;
}
```

```
int brojParticija(int n) {
    // particije broja n u kojima je najveći sabirak jednak n
    return brojParticija(n, n);
}

int main() {
    int n;
    cin >> n;
    cout << brojParticija(n) << endl;
}
```

Ako umesto uslova da su sabirci uređeni nerastuće stavimo uslov da su sabirci uređeni neopadajuće, tada uz broj čije particije tražimo šaljemo i broj smin **koji predstavlja donju granicu narednih sabiraka u particiji (svi sabirci moraju da imaju vrednost bar** smin**).**

- **Bazu čini slučaj kada je** n=0**, jer broj nula ima samo jednu particiju koja ne sadrži sabirke. Takođe, kada je** smin>n**, tada je broj particija nula, jer nijedna particija ne može da ima sabirak veći od zbira.**

- **Broj particija broja** n **koje sadrže** smin **jednak je broju particija vrednosti** n−smin **sa minimalnim sabirkom** smin**, dok je broj particija koje ne sadrže** smin **jednak broju particija broja** n **u kojima je najmanji sabirak** smin+1**. Dakle,** pn,smin=pn−smin,smin+pn,smin+1**.**

```
#include <iostream>
using namespace std;

// particije broja n u kojima je najveci sabirak jednak smax
int brojParticija(int n, int smin) {
    if (n == 0) return 1;
    if (smin > n) return 0;
    return  brojParticija(n, smin + 1) +
      brojParticija(n - smin, smin);
}

int brojParticija(int n) {
    // particije broja n u kojima je najmanji sabirak jednak 1
    return brojParticija(n, 1);
}

int main() {
    int n;
    cin >> n;
    cout << brojParticija(n) << endl;
}
```

Možemo formulisati još jedno rešenje u kome se broje neopadajuće sortirane particije u kojem u petlji razmatramo sve mogućnosti za vrednost narednog sabirka. Razmotrimo, na primer, particije broja 6**.**

```
1 1 1 1 1 1

1 1 1 1 2

1 1 1 3
```

1 1 2 2

1 1 4

1 2 3

1 5

2 2 2

2 4

3 3

6

Primetimo da je maksimalna vrednost prvog sabirka u svim particijama osim u onoj jednočlanoj jednakoj broju n**, manja ili jednaka** $n2$ **(u prethodnom primeru ni jedna particija ne počinje ni sa 4, ni sa 5). Zaista, ako bi prvi sabirak bio strogo veći od** $n2$ **i ako particija ne bi bila jednočlana i drugi sabirak bi morao biti strogo veći od** $n2$**, pa bi zbir bio strogo veći od** n**, što je nemouće. Stoga jednočlanu particiju posebno uračunavamo, a rekurzivno generišemo particije za sve moguće vrednosti prvog sabirka od** $smin$ **do** $\lfloor n2 \rfloor$**.**

```
#include <iostream>

using namespace std;

int brojParticija(int n, int smin) {
    if (n == 0)
        return 0;
    int br = 1;
    for(int i = smin; i <= n/2; i++)
        br += brojParticija(n - i, i);
    return br;
}

int brojParticija(int n) {
    return brojParticija(n, 1);
}

int main() {
    int n;
    cin >> n;
    cout << brojParticija(n) << endl;
    return 0;
}
```

Sva rešenja zasnovana na prostoj rekurziji su neefikasna, jer dolazi do ponavljanja identičnih rekurzivnih poziva i mogu se popraviti dinamičkim programiranjem. Prikažimo to na primeru brojanja nerastuće uređenih permutacija u kojima vršimo dva rekurzivna poziva.

Krenimo sa memoizacijom. Uvodimo matricu dimenzije $(n+1)\times(n+1)$ **koju popunjavamo vrednostima** -1**, čime označavamo da rezultat poziva funkcije još nije poznat. Pre nego što krenemo sa izračunavanjem proveravamo da**

li je u matrici vrednost različita od -1 i ako jeste, vraćamo tu upamćenu vrednost. Pre svake povratne vrednosti funkcije rezultat pamtimo u matrici.

```
#include <iostream>
#include <vector>
using namespace std;

int brojParticija(int n, int smax, vector<vector<int>>& memo) {
    if (memo[n][smax] != -1)
        return memo[n][smax];
    if (n == 0) return memo[n][smax] = 1;
    if (smax == 0) return memo[n][smax] = 0;
    int broj = brojParticija(n, smax-1, memo);
    if (n >= smax)
        broj += brojParticija(n-smax, smax, memo);
    return memo[n][smax] = broj;
}

int brojParticija(int n) {
    vector<vector<int>> memo(n + 1);
    for (int i = 0; i <= n; i++)
        memo[i].resize(n+1, -1);
    return brojParticija(n, n, memo);
}

int main() {
    int n;
    cin >> n;
    cout << brojParticija(n) << endl;
}
```

Umesto memoizacije možemo upotrebiti i dinamičko programiranje naviše. Prikažimo tabelu vrednosti funkcije za `n=7`**.**

n\smax	0	1	2	3	4	5	6	7
0	1	1	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1
2	0	1	2	2	2	2	2	2
3	0	1	2	3	3	3	3	3
4	0	1	3	4	5	5	5	5
5	0	1	3	5	6	7	7	7
6	0	1	4	7	9	10	11	11
7	0	1	4	8	11	13	14	15

Na osnovu baze indukcije znamo da će svi elementi prve vrste biti jednaki `1`**, a da će u prvoj koloni svi elementi osim početnog biti jednaki** `0`**. Jedan od načina da se matrica popunjava je postepeno uvećavajući vrednost** `n`**, tj. popunjavajući vrstu po vrstu.**

Element $p_{n,s}$ **zavisi od elemenata** $p_{n,s-1}$ **i (ako je** $n \geq s$**)** $p_{n-s,s}$ **i ako se vrste popunjavaju od gore naniže i sleva nadesno, prilikom njegovog izračunavanja oba elementa od kojih zavisi su već izračunata, što daje korektan algoritam.**

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

using namespace std;

int brojParticija(int N) {
    // alociramo matricu
    vector<vector<int>> dp(N+1);
    for (int n = 0; n <= N; n++)
        dp[n].resize(N+1);
    // popunjavamo prvu vrstu
    for (int smax = 0; smax <= N; smax++)
        dp[0][smax] = 1;
    // popunjavamo preostale elemente prve kolone
    for (int n = 1; n <= N; n++)
        dp[n][0] = 0;
    // popunjavamo jednu po jednu vrstu
    for (int n = 1; n <= N; n++)
        for (int smax = 1; smax <= N; smax++) {
            dp[n][smax] = dp[n][smax-1];
            if (n >= smax)
                dp[n][smax] += dp[n-smax][smax];
        }
    return dp[N][N];
}


int main() {
    int n;
    cin >> n;
    cout << brojParticija(n) << endl;
    return 0;
}
```

I vremenska i memorijska složenost prethodnog algoritma je $O(n^2)$ **iako se matrica popunjava vrstu po vrstu, to nije moguće popraviti (jer elementi zavise od elemenata koji se javljaju ne samo u prethodnoj, već i u ranijim vrstama, tako da je potrebno da istovremeno čuvamo sve prethodne vrste). Međutim, ako matricu popunjavamo kolonu po kolonu odozgo naniže, možemo dobiti memorijsku složenost** $O(n)$ **-- vremenska složenost ostaje** $O(n^2)$**. Naime, svaki element zavisi od elementa u istoj vrsti u prethodnoj koloni i elementa u istoj koloni u nekoj od prethodnih vrsta, tako da ako kolone popunjavamo odozgo naniže, možemo čuvati samo dve uzastopne kolone. Zapravo, možemo čuvati i samo jednu kolonu, ako njeno popunjavanje organizujemo tako da se tokom ažuriranja svi elementi pre tekuće vrste odnose na vrednosti tekuće kolone, a od tekuće vrste do kraja odnose na vrednosti prethodne**

kolone. Primetimo da se u delu gde je n<smax**, vrednosti između dve susedne kolone ne menjaju. Time dobijamo narednu optimizovanu implementaciju.**

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

using namespace std;

int brojParticija(int N) {
    vector<int> dp(N+1, 0);
    dp[0] = 1;
    for (int smax = 1; smax <= N; smax++)
        for (int n = smax; n <= N; n++)
            dp[n] += dp[n-smax];
    return dp[N];
}

int main() {
    int n;
    cin >> n;
    cout << brojParticija(n) << endl;
    return 0;
}
```

Dinamičkim programiranjem možemo optimizovati i slučaj kada se broje particije sa neopadajuće sortiranim sabircima. Prikažimo tabelu vrednosti funkcije za n=8**.**

n\smin	1	2	3	4	5	6	7	8
1	1	0	0	0	0	0	0	0
2	2	1	0	0	0	0	0	0
3	3	1	1	0	0	0	0	0
4	5	2	1	1	0	0	0	0
5	7	2	1	1	1	0	0	0
6	11	4	2	1	1	1	0	0
7	15	4	2	1	1	1	1	0
8	22	7	3	2	1	1	1	1

Tabelu možemo popunjavati vrstu po vrstu. Nažalost, memorijsku optitmizaciju je ovde teže ostvariti.

```
#include <iostream>

using namespace std;

const int MAX = 100;

int brojParticija(int N) {
    int br[MAX + 1][MAX + 1] = {0};

    // br[i][j] je broj particija broja i pomocu sabiraka >= smin
    for(int n = 1; n <= N; n++) {
        br[n][n] = 1;
        for(int smin = n-1; smin > 0; smin--)
            br[n][smin] = br[n][smin+1] + br[n-smin][smin];
```

```
    }
    return br[N][1];
}

int main() {
    int n;
    cin >> n;
    cout << brojParticija(n) << endl;
    return 0;
}
```

6. Svi podskupovi

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

Napiši program koji ispisuje sve podskupove datog skupa.

Ulaz

Sa standardnog ulaza se učitava broj n (važi $3 \leq n \leq 10$), a zatim n prirodnih brojeva, rastuće sortiranih, razdvojenih po jednim razmakom.

Izlaz

Na standardni izlaz ispisati sve podskupove učitanog skupa brojeva, svaki u posebnom redu, sa elementima razdvojenim jednim razmakom. Prvo se ređaju podskupovi u kojima prvi element nije uključen, a zatim oni u kojima jeste. U svakoj od te dve grupe, prvo se ispisuju podskupovi u kojima drugi element nije uključen, a zatim oni gde jeste i tako dalje.

Primer

Ulaz

3

1 2 3

Izlaz

3

2

2 3

1

1 3

1 2

1 2 3

Generisanje svih podskupova odgovara generisanju svih varijacija dužine n

od nula i jedinica (svaki element je ili uključen ili isključen), pa je rešenje slično onom prikazanom u zadatku Sve varijacije.

Iako mnogi savremeni jezici pružaju tip za reprezentovanje skupova, implementacija je jednostavnija i efikasnija ako se elementi skupa čuvaju u nizu. Da bismo izbegli potrebu za produžavanjem i skraćivanjem niza, niz možemo alocirati na maksimalnu moguću dužinu (broj elemenata polaznog skupa) i paralelno sa nizom možemo održavati broj elemenata podskupa koji je trenutno smešten u niz (on je skoro uvek strogo manji od dužine niza).

Ako je skup S prazan, onda je jedini njegov podskup prazan, a ako nije, onda se može razložiti na neki element x i skup S'=S\x dobijen kada se taj element izbaci iz polaznog skupa. Pošto je skup S' manji od skupa S, njegovi se podskupovi mogu odrediti rekurzivno. Svi podskupovi polaznog skupa S su onda oni koji su određeni za manji skup S', kao i svi oni koji se od njih dobijaju dodavanjem izdvojenog elementa x. Ovu konstrukciju nije ekonomično programski realizovati, jer se pretpostavlja da rezultat rada funkcije predstavlja skup svih podskupova skupa. Umesto takve funkcije definisaćemo proceduru koja neće istovremeno čuvati i vraćati sve podskupove već samo jedan po jedan nabrojati i obraditi. Do rešenja se može doći tako što se u rekurzivnom pozivu prosledi parcijalno popunjeni podskup koji ili ne sadrži ili sadrži izdvojeni element, a rekurzivni poziv ima zadatak da onda podskup koji je primio na sve moguće načine dopuni podskupovima smanjenog skupa S'.

Definišemo rekurzivnu funkciju koja na svakom narednom nivou rekurzije obrađuje naredni element polaznog skupa (predstavljenog nizom). U prvom slučaju ga ne dodaje u rezultujući podskup (takođe predstavljen nizom, koji prosleđujemo kao dodatni parametar) i prelazi na naredni nivo rekurzije, a u drugom ga dodaje na kraj trenutnog rezultujućeg podskupa i prelazi na naredni nivo rekurzije. Kada se ceo polazni niz iscrpi (kada je dubina rekurzije jednaka dužini polaznog niza), tada se trenutno akumulirani podskup ispisuje.

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

using namespace std;

void ispisi(const vector<int>& a, int n) {
  for (int k = 0; k < n; k++)
    cout << a[k] << " ";
  cout << endl;
}

void ispisi_sve_podskupove(const vector<int>& a, int i, vector<int>&
p, int j) {
  if (i == a.size())
    ispisi(p, j);
  else {
    ispisi_sve_podskupove(a, i + 1, p, j);
    p[j] = a[i];
```

```
    ispisi_sve_podskupove(a, i + 1, p, j + 1);
  }
}

void ispisi_sve_podskupove(const vector<int>& a) {
  vector<int> p(a.size());
  ispisi_sve_podskupove(a, 0, p, 0);
}

int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  ispisi_sve_podskupove(a);
  return 0;
}
```

Jedno rešenje je da se u posebnom nizu logičkih vrednosti nabrajaju sve varijacije skupa tačno-netačno. Svaka takva varijacija odgovara jednom podskupu, tako što se u podskup uključuju elementi sa onih pozicija na kojima je je vrednost tačno. Varijacije nabrajamo korišćenjem funkcije za određivanje sledeće varijacije, opisane u zadatku Sledeća varijacija.

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

using namespace std;

void ispisi(const vector<bool>& v, const vector<int>& a) {
  for (int i = 0; i < v.size(); i++)
    if (v[i])
      cout << a[i] << " ";
  cout << endl;
}

bool sledecaVarijacija(vector<bool>& v) {
  int i = v.size() - 1;
  while (i >= 0 && v[i])
    v[i--] = false;
  if (i < 0) return false;
  v[i] = true;
  return true;
}

void ispisiSvePodskupove(const vector<int>& a) {
  vector<bool> v(a.size(), false);
  do {
    ispisi(v, a);
  } while (sledecaVarijacija(v));
}

int main() {
  int n;
  cin >> n;
  vector<int> a(n);
```

```
  for (int i = 0; i < n; i++)
    cin >> a[i];
  ispisiSvePodskupove(a);
  return 0;
}
```

7.

Sledeći podskup

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

Napiši program koji određuje podskup skupa brojeva {1,…,n} koji u leksikografskom redosledu sledi neposredno iza datog podskupa. Podskupovi su zadati u obliku strogo rastuće sortiranih nizova.

Ulaz

Prva linija sadrži broj n(1≤n≤100), a naredna linija sadrži podskup čiji su elementi zadati sortirano rastuće, razdvojeni po jednim razmakom.

Izlaz

Na standardni izlaz u jednoj liniji ispisati elemente traženog podskupa tj. - ako je učitani podskup leksikografski najveći.

Primer

Ulaz

5

1 2 3 4 5

Izlaz

1 2 3 5

Napišimo, na primer, leksikografski uređen spisak svih podskupova skupa brojeva od 1 do 4.

-

1

12

123

1234

124

13

134

14

2

23

234

24

3

34

4

Možemo primetiti da postoje dva načina da se dođe do narednog podskupa. Analizirajmo ove skupove u istom redosledu, grupisane i na osnovu broja elemenata.

_ 1 12 123 1234

         124

     13 134

     14

  2 23 234

     24

  3 34

  4

Jedan način je *proširivanje* kada se naredni podskup dobija dodavanjem nekog elementa u prethodni. To su koraci u prethodnoj tabeli kod kojih se prelazi iz jedne u narednu kolonu. Da bi dobijeni podskup sledio neposredno iza prethodnog u leksikografskom redosledu, dodati element podskupu mora biti najmanji mogući. Pošto je svaki podskup sortiran, element mora biti za jedan veći od poslednjeg elementa podskupa koji se proširuje (izuzetak je prazan skup, koji se proširuje elementom 1). Jedini slučaj kada proširivanje nije moguće je kada je poslednji element podskupa najveći mogući (u našem primeru to je 4).

Drugi način je *skraćivanje* kada se naredni element dobija uklanjanjem nekih elemenata iz podskupa i izmenom preostalih elemenata. To su koraci u prethodnoj tabeli kod kojih se prelazi sa kraja jedne u narednu vrstu. U ovom slučaju skraćivanje funkcioniše tako što se iz podskupa izbaci završni najveći element, a zatim se najveći od preostalih elemenata uveća za 1 (on ne može biti najveći, jer su elementi unutar svakog podskupa strogo rastući). Ako nakon izbacivanja najvećeg elementa ostane prazan skup, naredna kombinacija ne postoji.

Podskupove možemo predstaviti dinamičkim nizom koji nam omogućva da elemente dodajemo i uklaljamo sa desnog kraja.

U jeziku C++ možemo upotrebiti vektor (kolekciju vector iz zaglavlja).

```
#include <iostream>

#include <vector>


using namespace std;
```

```
// na osnovu datog podskupa skupa {1, ..., n} određuje
// leksikografski naredni podskup i vraća da li takav
// podskup postoji
bool sledeciPodskup(vector<int>& podskup, int n) {
  // specijalni slučaj proširivanja praznog skupa
  if (podskup.empty()) {
    podskup.push_back(1);
    // podskup je uspešno pronađen
    return true;
  }
  // proširivanje
  if (podskup.back() < n) {
    // u podskup dodajemo element koji je za 1 veći od
    // trenutno najvećeg elementa
    podskup.push_back(podskup.back() + 1);
    // podskup je uspešno pronađen
    return true;
  }

  // skraćivanje
  // uklanjamo poslednji najveći element
  podskup.pop_back();
  // ako nema preostalih elemenata ne postoji naredni podskup
  if (podskup.empty())
    return false;
  // najveći od preostalih elemenata uvećavamo za 1
  podskup.back()++;
  // podskup je uspešno pronađen
  return true;
}

int main() {
  int n;
```

```
  cin >> n;
  vector<int> podskup;
  int x;
  while (cin >> x)
    podskup.push_back(x);
  if (sledeciPodskup(podskup, n)) {
    for (int x : podskup)
      cout << x << " ";
    cout << endl;
  } else {
    cout << "-" << endl;
  }
  return 0;
}
```

Podskupove možemo čuvati i u okviru niza koji je unapred alociran tako da može da smesti elemente najvećeg podskupa (onog koji ima tačno n elemenata). U tom slučaju je neophodno da održavamo i promenljivu u kojoj beležimo broj elemenata podskupa. Pošto se ona menja u funkciji koja određuje naredni podskup, potrebno je preneti je po referenci.

```
#include <iostream>

using namespace std;

// na osnovu datog podskupa skupa {1, ..., n} određuje
// leksikografski naredni podskup i vraća da li takav
// podskup postoji.
// Tekući podskup je smešten u nizu dužine k
bool sledeciPodskup(int podskup[], int& k, int n) {
  // specijalni slučaj proširivanja praznog skupa
  if (k == 0) {
    podskup[k++] = 1;
    return true;
  }

  // proširivanje
```

```
  if (podskup[k-1] < n) {
    // u podskup dodajemo element koji je za 1 veći od
    // trenutno najvećeg elementa
    podskup[k] = podskup[k-1] + 1;
    k++;
    return true;
  }

  // skraćivanje

  // izbacujemo najveći element iz podskupa
  k--;
  // ako nema preostalih elemenata, naredni podskup ne postoji
  if (k == 0)
    return false;

  // najveći od preostalih elemenata uvećavamo za 1
  podskup[k-1]++;
  return true;
}


int main() {
  int n;
  cin >> n;
  // elementi podskupa
  const int MAX = 100;
  int podskup[MAX];
  // broj elemenata podskupa
  int k = 0;
  // ucitavamo elemente datog podskupa
  int x;
  while (cin >> x)
    podskup[k++] = x;
```

```
  // pokusavamo da pronadjemo naredni podskup
  if (sledeciPodskup(podskup, k, n)) {
    // ako postoji, ispisujemo ga
    for (int i = 0; i < k; i++)
      cout << podskup[i] << " ";
    cout << endl;
  } else
    // naredni podskup ne postoji
    cout << "-" << endl;
  return 0;
}
```

6.

Svi binarni nizovi bez susednih jedinica

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

Napiši program koji ispisuje sve nizove binarnih brojeva date dužine u kojima se ne javljaju dve uzastopne jedinice. Brojeve ispisati u leksikografskom redosledu.

Ulaz

Sa standardnog ulaza se unosi broj n.

Izlaz

Na standardni izlaz ispisati tražene brojeve, svaki u posebnom redu.

Primer

Ulaz

3

Izlaz

000

001

010

100

101

Zadatak predstavlja modifikaciju zadatka Sve varijacije.

Jedan način je da definišemo rekurzivnu funkciju koja generiše tražene brojeve. Ona dobija prefiks dužine i

i pokušava na sve načine da ga proširi (kroz rekurziju proširujemo niz karaktera u startu alociran na dužinu n dužinu i njegovog popunjenog dela). Ako je i=n, tada je ceo niz popunjen i ispisuje se. U suprotnom, na poziciju i uvek možemo dopisati nulu i rekurzivno nastaviti sa produžavanjem tako dobijene niske. Sa druge strane, jedinicu možemo upisati samo ako prethodni karakter nije jedinica (u suprotnom bismo dobili dve uzastopne jedinice). To se dešava ili kada nema prethodne cifre (kada je i=0) ili kada je prethodna cifra (na poziciji i−1

) različita od jedinice.

```
#include <iostream>

#include <string>


using namespace std;


void obradi(const string& binarni) {

  cout << binarni << endl;

}


void obradiSveBinarneBez11(string& binarni, int i) {

  if (i == binarni.size())

    obradi(binarni);

  else {

    binarni[i] = '0';

    obradiSveBinarneBez11(binarni, i+1);

    if (i == 0 || binarni[i-1] != '1') {

      binarni[i] = '1';

      obradiSveBinarneBez11(binarni, i+1);

    }

  }

}


void obradiSveBinarneBez11(int n) {

  string binarni(n, '0');

  obradiSveBinarneBez11(binarni, 0);

}
```

```
int main() {
  int n;
  cin >> n;
  obradiSveBinarneBez11(n);
  return 0;
}
```

Jedna mogućnost je da se upotrebi funkcija koja generiše narednu u leksikografskom poretku binarnu nisku bez susednih jednica (ta funkcija je opisana u zadatku Sledeći binarni niz bez susednih jedinica). Kreće se od niske koja sadrži n

nula i ispisuje se i računa naredna varijacija sve dok takva postoji.

```
#include <iostream>
#include <string>

using namespace std;

void obradi(const string& binarni) {
  cout << binarni << endl;
}

bool sledeciBinarniBez11(string& s){
  int n = s.length();
  int i = n - 1;
  while ((i >= 0  && s[i] == '1') ||
         (i > 0 && s[i - 1] == '1'))
    s[i--] = '0';
  if (i < 0)
    return false;
  s[i] = '1';
  return true;
}

void obradiSveBinarneBez11(int n) {
  string binarni(n, '0');
  do {
```

```
    obradi(binarni);
  } while (sledeciBinarniBez11(binarni));
}



int main() {
  int n;
  cin >> n;
  obradiSveBinarneBez11(n);
  return 0;
}
```

8.

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

Napisati program kojim se prikazuju dekadni zapisi svih prirodnih brojeva koji u binarnom sistemu imaju najviše n binarnih cifara i nemaju dve uzastopne nule.

Ulaz

Prva linija standardnog ulaza sadrži prirodan broj n(1≤n≤20).

Izlaz

Na standardnom izlazu prikazati tražene brojeve u rastućem poretku, svaki broj u posebnoj liniji.

Primer

Ulaz

3

Izlaz

1

2

3

5

6

7

vaj zadatak je veoma sličan zadatku Svi binarni nizovi bez susednih jedinica i može se rešavati analogno njemu. Osnova razlika je to što je

nakon generisanja binarnog niza potrebno izračunati njegovu dekadnu vrednost. To možemo uraditi Hornerovom šemom, kako je prikazano u zadatku Broj formiran od datih cifra s leva na desno.

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

using namespace std;

int binUDekadni(const vector<bool>& b) {
  int d = 0;
  for (bool x : b)
    d = 2 * d + (x ? 1 : 0);
  return d;
}

void generisi_(vector<bool>& tekuci, int poz) {
  if (poz == tekuci.size()) {
    cout << binUDekadni(tekuci) << endl;
    return;
  }
  if (poz > 0 && tekuci[poz-1] != false) {
    tekuci[poz] = false;
    generisi_(tekuci, poz + 1);
  }
  tekuci[poz] = true;
  generisi_(tekuci, poz + 1);
}

void generisi(int n) {
  for (int brojBinCifara = 1; brojBinCifara <= n; brojBinCifara++) {
    vector<bool> tekuci(brojBinCifara);
    generisi_(tekuci, 0);
  }
}
```

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

Još jedna mogućnost je da umesto u obliku niza cifara (niza podataka tipa char, int ili bool) broj pamtimo u obliku jednog celobrojnog podatka (najbolje tipa unsigned), čiji interni binarni zapis predstavlja naš tekući niz cifara. Proveru poslednje cifre možemo izvršiti bitovskom konjunkcijom (&) sa brojem 1 (poslednja cifra je 1 ako i samo ako se dobije rezultat 1, tj. vrednost različita od 0), dodavanje nule na desni kraj možemo ostvariti pomeranjem (šiftovanjem) ulevo za jedno mesto (operatorom <<), a dodavanje jedinice možemo ostvariti šiftovanjem ulevo za jednom mesto i zatim izračunavanjem bitovske disjunkcije (|) sa brojem 1. Ovim se izbegava potreba za preračunavanjem niza cifara u dekadni sistem (ispisivanjem podatka tipa unsigned ispisuje se automatski njegova dekadna vrednost). Naglasimo i da je funkciju generisanja potrebno posebno pozivati za svaki broj cifara od 1 pa do unetog maksimalnog broja cifara n.

```
#include <iostream>

using namespace std;

void generisi_(unsigned tekuci, int preostaloCifara) {
  if (preostaloCifara == 0) {
    cout << tekuci << endl;
    return;
  }
  if (tekuci & 1 != 0)
    generisi_(tekuci << 1, preostaloCifara - 1);
  generisi_((tekuci << 1) | 1, preostaloCifara - 1);
}

void generisi(int n) {
  for (int brojBinCifara = 1; brojBinCifara <= n; brojBinCifara++)
    generisi_(0, brojBinCifara);
```

```
}


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

8.

Sve kombinacije

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

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, 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, međutim, 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},tadanepostojivrednostkojujemogućestavitinapozicijui,pamožemoizaći izrekurzije,jersetrenutnakombinacijanemožepopunitidokraja.Usuprotnommo žemorazmotritidvemogućnosti.Prvonapozicijuimožemopostavitielementn*{min}* *irekurzivnoizvršitipopunjavanjenizaodpozicijei+1,adrugomožemotajelement preskočitiirekurzivnompozivuponovozahtevatidasepopunipozicijai.Uobaslu čajaseskupelemenatasužavana*

{n{min}+1, \ldots, n_{max}}$.

Pretragu možemo saseći i malo ranije. Naime, pošto su ponavljanja zabranjena kada je broj elemenata tog skupa (a to je n−nmin+1

) manji od broja preostalih pozicija koje treba popuniti (a to je k−i

), već tada možemo saseći pretragu, jer ne postoji mogućnost da se kombinacija uspešno dopuni 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 zadatku Sledeća 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 kombi=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 komb1=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 kombi<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;
}
```