Глава 8

Динамичко програмирање

8.1 Појам и облици динамичког програмирања

У многим случајевима се дешава да током извршавања рекурзивне функције долази до *преклапања рекурзивних позива* (енгл. overlapping recursive calls) тј. да се идентични рекурзивни позиви (рекурзивни позиви са идентичним параметрима) извршавају више пута. Ако се то дешава често, програми су по правилу веома неефикасни (у многим случајевима број рекурзивних позива, па самим тим и сложеност бива експоненцијална у односу на величину улаза). До ефикаснијег решења се често може доћи техником **динамичког програмирања**. Оно често временску ефикасност поправља ангажовањем додатне меморије у којој се бележе резултати извршених рекурзивних позива. Динамичко програмирање долази у два облика.

- Техника **мемоизације** или **динамичког програмирања наниже** задржава рекурзивну дефиницију али у додатној структури података (најчешће низу или матрици, ређе мапи тј. речнику) бележи све резултате рекурзивних позива, да бих их у наредним позивима у којима су параметри исти само очитала из те структуре.
- Техника **динамичког програмирања навише** у потпуности уклања рекурзију и ту помоћну структуру података попуњава исцрпно у неком систематичном редоследу. Дакле, рекурзивна конструкција се замењује индуктивном, тј. итеративном.

Док се код мемоизације може десити да се рекурзивна функција не позива за неке вредности параметара, код динамичког програмирања навише се израчунавају вредности функције за све могуће вредности њених параметара мањих од вредности која се заправо тражи у задатку. Иако се на основу овога може помислити да је мемоизација ефикаснија техника, у пракси је чешћи случај да је током одмотавања рекурзије потребно израчунати вредност рекурзивне функције за баш велики број различитих параметара, тако да се ове предности мемоизаиције у пракси ретко сређе.

Најбољи начин да разјаснимо технику динамичког програмирања је да је илуструјемо на низу погодно одабраних примера. Кренућемо од Фибоначијевог низа, који је опште познати проблем и кроз чије се решавање могу илустровати већина основних концепата динамичког програмирања.

Задатак: Пчеле и трутови

Пчела матица носи јајашца. Ако трут оплоди јајашце пчеле, тада се из њега рађа женска пчела. Ако се јајашце не оплоди, онда се из њега излеже трут. Дакле, женска пчела има два родитеља, док трут има само једног (он нема оца, већ само мајку). Пчела има две баке (мамину и татину маму) и једног деду (маминог тату), док трут има једну баку и једног деду (мамине родитеље). Напиши програм који одређује колико предака у некој генерацији има трут.

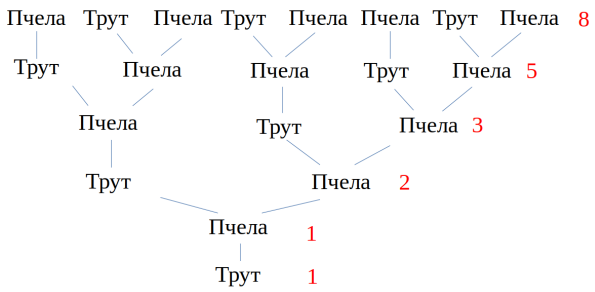
Улаз: Са стандардног улаза се уноси број n ($1 \leq n \leq 50$) који означава редни број генерације: 0 је генерација самог трута, 1 је генерација његове мајке, 2 је генерација његове баке и деде и тако даље у прошлост.

Излаз: На станадрни излаз исписати укупан број предака трута у генерацији n.

Пример

Улаз	*Излаз*
5	8

Објашњење



Слика 8.1: Породично стабло једног трута

Решење

Обележимо са f_n^m број мушких предака које трут има у генерацији n и f_n^z број женских предака које трут има у генерацији n. Важи да је $f_0^m = 1$ и $f_0^z = 0$ (у нултој генерацији је само трут). За свако $i \geq 0$ важи $f_{i+1}^m = f_i^z$, јер само женске јединке имају очеве, док је $f_{i+1}^z = f_i^m + f_i^z$, јер и мушке и женске јединке имају мајке.

Фибоначијев низ

Уместо два, можемо доћи и до једног рекурентног низа на основу којег добијамо решење. Обележимо са $f_n = f_n^m + f_n^z$ укупан број предака трута у генерацији n. Пошто је $f_0^m = f_1^z = 1$ и $f_0^z = f_1^m = 0$, важи да је $f_0 = f_1 = 1$. За свако $i \geq 0$ важи да је $f_{i+1}^z = f_i^m + f_i^z = f_i$. Зато за свако $i \geq 0$ важи да је $f_{i+2}^m = f_{i+1}^z = f_i$. Зато за свако $i \geq 0$ важи $f_{i+2} = f_{i+2}^m + f_{i+2}^z = f_i + f_{i+1}$. Дакле, важи следеће:

$$f_0 = f_1 = 1 \qquad f_{i+2} = f_{i+1} + f_i.$$

На основу овога можемо закључити да број предака задовољава услове чувеног Фибоначијевог низа бројева у којем је сваки наредни елемент једнак збиру претходна два (елементи тог низа су $1, 1, 2, 3, 5, 8, 13, 21, \ldots$).

Основна рекурзивна имплементација

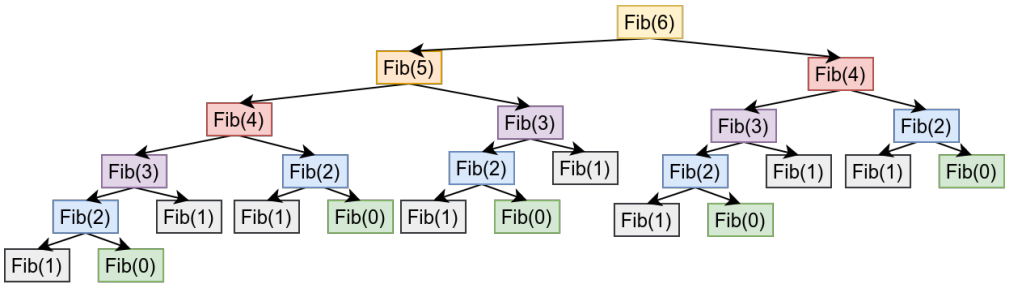
Имплементацију можемо направити рекурзивно, директно на основу дефиниције (ако је $n = 0$ или $n = 1$ функција враћа 1, а у супротном враћа збир резултата рекурзивних позива за вредности $n - 1$ и $n - 2$).

```
long long f(int n) {
  if (n == 0 || n == 1)
    return 1;
  return f(n-1) + f(n-2);
}
```

Директна рекурзивна имплементација је типичан пример неефикасне имплементације јер се рекурзивни позиви за исте вредности параметара понављају више пута. Ако рекурзивну функцију модификујемо тако да на почетку свог извршавања исписује број n, за позив `fib(6)` добијамо следећи испис.

```
6 5 4 3 2 1 0 1 2 1 0 3 2 1 0 1 4 3 2 1 0 1 2 1 0
```

Рекурзивни позиви се могу представити и дрветом.



Слика 8.2: Дрво рекурзивних позива

Позив `fib(6)` врши се један пут, `fib(5)` један пут, `fib(4)` два пута, `fib(3)` три пута, `fib(2)` пет пута, `fib(1)` осам пута и `fib(0)` пет пута. Приметимо да број позива одговара члановима Фибоначијевог низа.

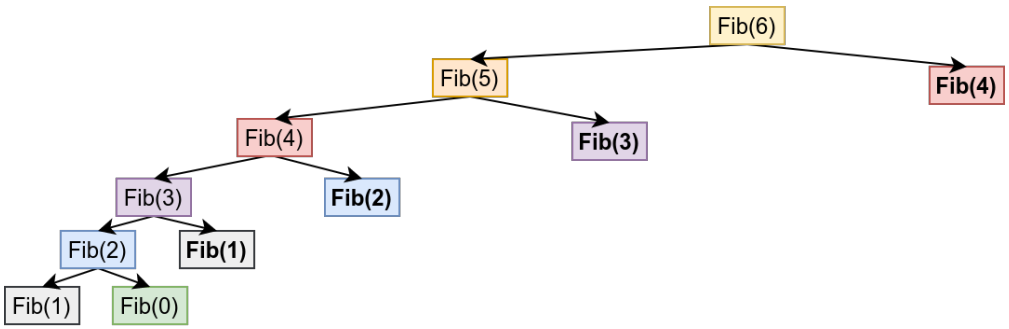
Анализа сложености. Рекурзивна имплементација задовољава једначину $T(n) = T(n-1)+T(n-2)+O(1)$, док је $T(1) = T(0) = O(1)$. Решење ове нехомогене једначине једнако је збиру решења њеног хомогеног дела и неког партикуларног решења, при чему је решење хомогеног дела $F(n) = F(n-1) + F(n-2)$ баш Фибоначијев низ, чије решење се експлицитно може изразити тзв. Бинеовом формулом помоћу $F(n) = \frac{\phi^n - \psi^n}{\phi - \psi}$, где је $\phi = \frac{1+\sqrt{5}}{2}$ и $\psi = \frac{1-\sqrt{5}}{2}$ (напоменимо и да се Бинеова формула може употребити за ефикасно израчунавање чланова Фибоначијевог низа). Решење полазне једначине је $T(n) = O(\frac{1+\sqrt{5}}{2})^n$, тако да је решење експоненцијалне сложености, што је јако неефикасно.

Стога, да би било могуће вршити израчунавање и за веће вредности n, потребно је убрзати имплементацију коришћењем техника динамичког програмирања.

Мемоизација уз коришћење асоцијативног низа

Једна могућност је да се примени мемоизација. Резултате свих рекурзивних позива ћемо памтити у некој помоћној структури података и на почетку сваког рекурзивног позива ћемо претрагом те структуре проверавати да ли је резултат за текућу вредност параметра можда већ израчунат раније. Пошто вредности параметара треба да пресликамо у резултате рекурзивних позива треба да користимо неки облик пресликавања кључева у вредности. То може бити асоцијативни низ, (мапа, односно речник).

Анализа сложености. На овај начин добијамо алгоритам чија је и временска и меморијска сложеност $O(n)$, јер се за свако n израчунавање врши само једном. Дрво рекурзивних позива у овом случају је приказано на слици (спустом дуж леве гране рачунају се вредности за све параметре, док се онда свака од десних грана сасеца јер је резултат већ од раније познат).



Слика 8.3: Дрво рекурзивних позива уз мемоизацију

```
long long fib(int n, unordered_map<int, long long>& memo) {
  // ako je vrednost za parametar n već računata
  // vraćamo ranije izračunatu vrednost
  auto it = memo.find(n);
  if (it != memo.end())
     return it->second;
  // pre nego što vratimo vrednost, pamtimo je u nizu
```

```
  if (n == 0) return memo[n] = 1;
  if (n == 1) return memo[n] = 1;
  return memo[n] = fib(n-1, memo) + fib(n-2, memo);
}

long long fib(int n) {
  // mapa u kojoj se vrednostima parametara rekurzivnog poziva
  // pridružuju vrednosti rekurzivnog poziva
  unordered_map<int, long long> memo;
  return fib(n, memo);
}
```

Мемоизација уз коришћење класичног низа

Иако мапа тј. речник представља природан избор за чување коначног пресликавања, њена употреба у служби мемоизације није честа. Наиме, показује се да се боље перформансе постижу ако се уместо мапе употреби низ (било статички, било динамички алоциран). Тиме се може ангажовати мало више меморије у односу на коришћење асоцијативног низа, међутим, претрага и упис вредности су донекле бржи. У ситуацијама у којима се вредност израчунава за велики број улазних параметара (а код Фибоначија можемо бити сигурни да се приликом израчунавања вредности за параметар n врше позиви за све вредности од 0 до $n-1$), низ може бити чак и меморијски ефикаснији у односу на мапу. Стога се у склопу динамичког програмирања обично користите низови и матрице.

Речник ћемо реализовати помоћу низа тако што ћемо на месту i памтити вредност позива за вредност параметра i. Потребно је још некако обележити вредности параметара у низу за које још не знамо резултате рекурзивних позива. За то се обично користи нека специјална вредност. Ако знамо да ће сви резултати бити ненегативни бројеви, можемо употребити, на пример, -1, а ако знамо да ће бити позитивни бројеви, можемо употребити, на пример 0. Ако немамо таквих претпоставки можемо ангажовати додатни низ логичких вредности којима ћемо експлицитно кодирати да ли за неки параметар знамо или не знамо вредност. Пошто смо сигурни да ће током рекурзије све вредности параметара позитивне и да је највећа вредност која се може јавити као параметар вредност иницијалног позива n, довољно је да алоцирамо низ величине $n+1$.

```
long long fib(int n, vector<long long>& memo) {
  // ako je vrednost za parametar n već računata
  // vraćamo ranije izračunatu vrednost
  if (memo[n] != -1)
    return memo[n];
  // pre nego što vratimo vrednost, pamtimo je u nizu
  if (n == 0) return memo[n] = 1;
  if (n == 1) return memo[n] = 1;
  return memo[n] = fib(n-1, memo) + fib(n-2, memo);
}

long long fib(int n) {
  // alociramo niz veličine n+1 i popunjavamo ga vrednostima -1
  // kojima označavamo da ta vrednost još nije računata
  vector<long long> memo(n+1, -1);
  return fib(n, memo);
}
```

Динамичко програмирање навише

Могуће је применити и динамичко програмирање навише. Техника динамичког програмирања навише подразумева да се уклони рекурзија и да се све вредности у низу попуне неким редоследом. Пошто вредности на вишим позицијама зависе од оних на нижим, низ попуњавамо слева надесно. На прва два места уписујемо нулу и јединицу, а затим у петљи сваку наредну вредност израчунавамо на основу две претходне. На крају враћамо тражену вредност на последњој позицији у низу.

```
vector<long long> dp(n + 1);
dp[0] = dp[1] = 1;
```

```
for (int i = 2; i <= n; i++)
  dp[i] = dp[i-1] + dp[i-2];
cout << dp[n] << endl;
```

Динамичко програмирање навише - меморијска оптимизација

Једноставно се примећује да вредност наредног члана Фибоначијевог низа зависи само од вредности његова претходна два члана. Зато можемо направити меморијску оптимизацију и не морамо истовремено чувати све чланове у низу.

```
long long fpp = 1;
long long fp = 1;
for (int i = 2; i <= n; i++) {
  long long f = fp + fpp;
  fpp = fp;
  fp = f;
}

cout << fp << endl;
```

Анализа сложености. Меморијска сложеност овог решења је $O(1)$, док је временска сложеност $O(n)$.

"Рецепт" за динамичко програмирање

Низ корака који смо применили у овом задатку јављаће се веома често.

1. Индуктивно-рекурзивном конструкцијом конструише се рекурзивна дефиниција која је неефикасна јер се исти позиви прекапају тј. функција се за исте аргументе позива више пута.
2. Техником мемоизације побољшава се сложеност тако што се у помоћном речнику (најчешће имплементираном помоћу низа или матрице) чувају израчунати резултати рекурзивних позива.
3. Уместо технике мемоизације која је вођена рекурзијом и у којој се вредности попуњавају по потреби, рекурзија се може елиминисати и речник (низ тј. матрица) се може цео попунити (неким редоследом).
4. Често је могуће да се изврши меморијска оптимизација на основу тога што се примећује да након попуњавања одређених елемената низа тј. матрице неке вредности (ранији елементи, раније врсте или колоне) више нису потребне, тако да се уместо истовремног памћења свих елемената памти само неколико претходних (они који су потребни за даље попуњавање).

Два рекурентна низа

Имплементацију је могуће направити и коришћењем засебних низова који описују број мушких тј. женских предака. Подсетимо се, важи, $f_0^m = 1, f_0^z = 1$, и за свако $i \geq 0$ важи $f_{i+1}^m = f_i^z$, као и $f_{i+1}^z = f_i^m + f_i^z$. Ове једнакости нам дају две узајамно рекурентно дефинисане серије бројева, на основу чега можемо направити било рекурзивну, било итеративну имплементацију. Коначно решење представља збир $f_n^m + f_n^z$.

У рекурзивној имплементацији је потребно имплементирати две узајамно рекурзивне функције. У језику C++ је потребно да је компилатор познаје сваку од функција пре позива, па је стога на почетку добро декларисати тј. навести прототипове обе функције (мада би довољно било декларисати само другу од њих).

И у оваквој рекурзивној имплементацији рекурзивни позиви за исте вредности улазних параметара се понављају више пута и стога је она прилично неефикасна.

```
// broj muskih jedinki u generaciji n
long long fm(int n);
// broj zenskih jedinki u generaciji n
long long fz(int n);

// broj muskih jedinki u generaciji n
long long fm(int n) {
  // u generaciji 0 postoji samo trut
  if (n == 0)
    return 1;
  // samo zenke iz naredne generacije imaju oceve
```

```
  return fz(n-1);
}

// broj zenskih jedinki u generaciji n
long long fz(int n) {
  // u generaciji 0 postoji samo trut
  if (n == 0)
    return 0;
  // i muzjaci i zenke iz naredne generacije imaju majke
  return fm(n-1) + fz(n-1);
}

// ukupan broj jedinki u generaciji n
long long f(int n) {
  // sabiramo muske i zenske jedinke u generaciji n
  return fm(n) + fz(n);
}
```

Наравно, и у овој имплементацији је могуће применити динамичко програмирање и у крајњој инстанци добити наредну итеративну имплементацију.

У итеративној имплементацији одржавамо две променљиве у којима чувамо текући број мушких и женских јединики у генерацији (иницијализујемо их на 1 и 0, на основу нулте генерације у којој постоји само трут). Затим n пута ажурирамо вредности на основу изведених рекурентних веза. Потребно је једино обратити пажњу да се ажурирање друге променљиве врши увек на основу обе старе вредности, а не на основу ажуриране вредности прве променљиве (за то је потребно употребити помоћну променљиву).

```
int n;
cin >> n;
// broj muskih i zenskih jedinki
// u generaciji 0 postoji samo trut
long long fm = 1, fz = 0;
// n puta azuriramo brojeve prelazeci sa tekuce na prethodnu
// generaciju
for (int i = 1; i <= n; i++) {
  // samo zenke imaju oceve, dok majke imaju i muzjaci i zenke
  long long fm_ = fz, fz_ = fm + fz;
  fm = fm_; fz = fz_;
}
// ukupan broj jedinki u generaciji n
cout << fm + fz << endl;
```

Могуће је дефинисати и репно-рекурзивну функцију која ефикасно израчунава решење.

```
// izracunaj ukupan broj jedinki u generaciji n, ako znas da u
// generaciji i postoji fm muskih i fz zenskih jedinki
long long f_(long long fm, long long fz, int i, int n) {
  // ovo je bas generacija n - izracunaj i vrati ukupan broj jedinki
  if (i == n)
    return fm + fz;
  // mozes da izracunas broj muskih i zenskih jedinki u generaciji i+1
  return f_(fz, fm+fz, i+1, n);
}

// izracunava broj jediniki u generaciji n
long long f(int n) {
  // u generaciji 0 postoji jedna muska i nula zenskih jediniki -
  // kreni od toga
  return f_(1, 0, 0, n);
}
```

8.2 Бројање комбинаторних објеката

Једна веома значајна примена технике динамичког програмирања је у проблемима пребројавања.

Задатак: Број комбинација

Напиши програм који одређује број комбинација без понављања дужине k из скупа од k елемената (тј. број различитих комбинација у игри лото ако се из бубња који садржи n лоптица извлачи њих k).

Улаз: Са стандардног улаза се извлачи број k ($1 \leq k \leq n$) и број n ($1 \leq n \leq 40$).

Излаз: На стандардни излаз исписати тражени број комбинација.

Пример 1

Улаз	*Излаз*
3 5	10

Објашњење

То су комбинације $(1, 2, 3)$, $(1, 2, 4)$, $(1, 2, 5)$ $(1, 3, 4)$, $(1, 3, 5)$, $(1, 4, 5)$, $(2, 3, 4)$, $(2, 3, 5)$, $(2, 4, 5)$ и $(3, 4, 5)$.

Пример 2

Улаз

```
7
39
```

Излаз

```
15380937
```

Решење

Иако постоје разни начини да се до решења овог задатка дође, приказаћемо технику засновану на томе да програм који набраја све комбинаторне објекте мало по мало трансформишемо до ефикасног прорама који их броји. Ова техника није специфична за комбинације без понављања и може се применити на бројање било које врсте комбинаторних објеката које набрајамо рекурзивном функцијом.

Рецимо да је број комбинација једнак биномном коефицијенту

$$\binom{n}{k} = \frac{n!}{(n-k)!k!},$$

међутим израчунавање на основу директне примене ове формуле би веома брзо довело до прекорачења (услед веома брзог раста факторијелске функције). Мало боља ситуација је да се израчуна

$$\binom{n}{k} = \frac{n(n-1)\ldots(n-k+1)}{k!},$$

но ни то у потпуности не уклаља проблем прекорачења, јер именилац може бити превелики.

Рекурзивна функција која израчунава број комбинација

Можемо кренути од процедуре за генерисање свих комбинација у којој се одржава интервал $[n_{min}, n_{max}]$ из којег се могу узети вредности којима се проширује започета комбинација и у којој се кроз два рекурзивна позива разматра могућност да се вредност n_{min} уврсти у комбинацију и могућност да се она не уврсти у комбинацију. Та је процедура објашњена у задатку Све комбинације,

Уместо процедуре која исписује комбинације, дефинишемо фукцију која враћа број комбинација. Пошто нам је битан само број комбинација, а не и саме комбинације, можемо у потпуности избацити низ који се попуњава и уместо њега прослеђивати само његову дужину k.

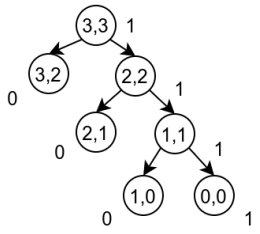
```
long long brojKombinacija(int i, int k,
                          int n_min, int n_max) {
  // ako je popunjen ceo niz postoji jedna kombinacija
  if (i == k) return 1;
  // ako niz nije moguće popuniti do kraja, tada nema kombinacija
  if (k - i > n_max - n_min + 1)
    return 0;
  // broj kombinacija je jednak zbiru kombinacija u dva slučaja
  return brojKombinacija(i+1, k, n_min+1, n_max) +
         brojKombinacija(i, k, n_min+1, n_max);
}

long long brojKombinacija(int K, int N) {
  return brojKombinacija(0, K, 1, N);
}
```

Можемо приметити да нам конкретне вредности k и i нису битне, већ је битан само број елемената у интервалу $[i, k)$ тј. разлика $k-i$. Слично, нису нам битне ни конкретне вредности n_{max} и n_{min} већ само број елемената у сегменту $[n_{min}, n_{max}]$ тј. вредност $n_{max} - n_{max} + 1$. Ако те две величине заменимо са k тј. n добијамо наредну дефиницију.

```
long long brojKombinacija(int k, int n) {
  // ako je popunjen ceo niz postoji jedna kombinacija
  if (k == 0) return 1;
  // ako niz nije moguće popuniti do kraja, tada nema kombinacija
  if (k > n) return 0;
  // broj kombinacija je jednak zbiru kombinacija u dva slučaja
  return brojKombinacija(k-1, n-1) + brojKombinacija(k, n-1);
}
```

Ако функцију позовемо за вредности $k \leq n$, случај $k > n$ може наступити једино из другог рекурзивног позива за $k = n$ (јер однос између k и n у првом рекурзивном позиву остаје непромењен, а у другом се мења само за 1). Међутим, како је илустровано на наредној слици, у случају позива функције за $k = n$ добиће се увек повратна вредност 1 (један рекурзивни позив ће увек враћати нулу, а други ће проузроковати смањивање оба аргумента све док се не дође до $k = n = 0$, када ће се 1 вратити на основу првог излаза из рекурзије), што је сасвим у складу са тим да тада постоји само једна комбинација.



Слика 8.4: Израчунавање коефицијената за $k = n$

На основу овога из рекурзије можемо изаћи за $k = n$ вративши вредност 1, чиме онда елиминишемо потребу за провером да ли је $k > n$ (наравно, под претпоставком да ћемо функцију позивати само за $k \leq n$).

Примећујемо да смо овом трансформацијом добили чувене особине биномних коефицијената.

$$\binom{n}{0} = 1, \qquad \binom{n}{n} = 1, \qquad \binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.$$

Оне чине основу Паскаловог троугла у ком се налазе биномни коефицијенти (у наредној табели коефицијент $\binom{n}{k}$ је написан у врсти n и колони k, да би се приказао уобичајени облик Паскаловог троугла, који се попуњава врсту по врсту).

```
1                      (0,0)
1  1                   (1,0) (1,1)
1  2  1                (2,0) (2,1) (2,2)
1  3  3  1             (3,0) (3,1) (3,2) (3,3)
1  4  6  4  1          (4,0) (4,1) (4,2) (4,3) (4,4)
1  5 10 10  5  1       (5,0) (5,1) (5,2) (5,3) (5,4) (5,5)
1  6 15 20 15  6  1    (6,0) (6,1) (6,2) (6,3) (6,4) (6,5) (6,6)
```

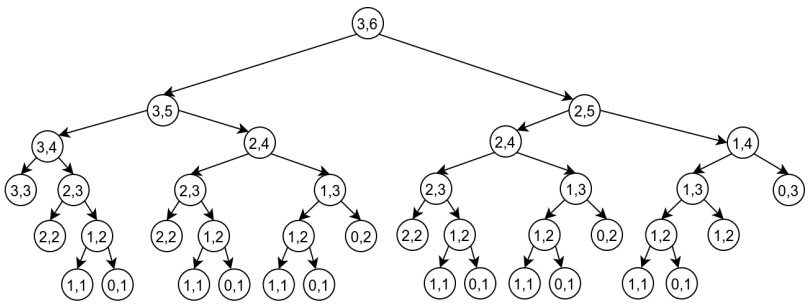
Прва веза говори да су елементи прве колоне увек једнаки 1, друга да су на крају сваке врсте елементи такође једнаки 1, а трећа да је сваки елемент у троуглу једнак збиру елемента непосредно изнад њега и елемента непосредно испред тог.

Наравно, до ових формула и до рекурзивне дефиниције смо могли доћи и директно, разматрањем дефиниција комбинација, на пример, у моделу где се k куглица без враћања извлаче из бубња у ком се налази n различитих куглица. Постоји јединствен начин да се из бубња не извуче ни једна куглица. Такође, постоји јединствен начин да се из бубња извуче свих n куглица. У супротном (ако је $0 < k < n$), тада све начине раздвајамо на оне у којима јесте и на оне у којима није извучена прва куглица (куглица са најмањим бројем). Ако она јесте извучена, преостало је да се извуче још $k-1$ куглица из бубња у ком се налази $n-1$ куглица, а ако није, тада је преостало да се извуче још k куглица из бубња у ком се налази још $n-1$ куглица (пошто смо претпоставили да у другој групи начина куглица са најмањим бројем неће бити међу извученим куглицама, можемо је одмах избацити из бубња и склонити негде са стране).

```
long long brojKombinacija(int k, int n) {
  // ako je popunjen ceo niz postoji jedna kombinacija
  if (k == 0) return 1;
  // ako treba popuniti još tačno n elemenata, tada postoji
  // tačno jedna kombinacija
  if (k == n) return 1;
  // broj kombinacija je jednak zbiru kombinacija u dva slučaja
  return brojKombinacija(k-1, n-1) + brojKombinacija(k, n-1);
}
```

Анализа сложености. Време које се утроши у сваком рекурзивном позиву (не рачунајући рекурзивне позиве који се из њега из позивају) је очигледно $O(1)$. Једначина којом се описује време рада функције је $T(k,n) = T(k-1,n-1) + T(k,n-1) + O(1)$, $T(0,n) = T(n,n) = O(1)$, и време извршавања је $T(k,n) = O(\binom{n}{k})$. За фиксирано k, ово је сложеност описана полиномом променљиве n, који може бити веома високог степена (k), док за фиксирано n овај број расте експоненцијално са порастом k. У сваком случају, јасно је да је сложеност изузетно висока и да је овај програм практично употребљив за веће вредности n и k.

Разлог овој неефикасности су поновљени рекурзивни позиви, што се може видети на слици. Сваком листу дрвета одговара тачно једна комбинација, па пошто укупно комбинација има $\binom{n}{k}$, укупан број рекурзивних позива је ограничен са $2\binom{n}{k}$ (јер у бинарном дрвету не може бити више него дупло више чворова него листова). У сваком позиву се врши $O(1)$ операција, па је сложеност $O(\binom{n}{k})$.



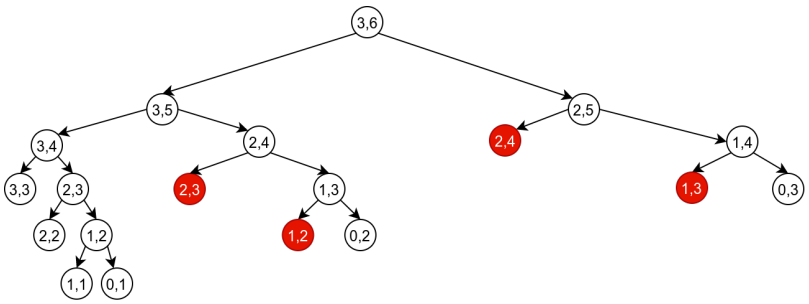
Слика 8.5: Дрво рекурзивних позива - сваки лист одговара тачно једној комбинацији, а приметно је понављање идентичних рекурзивних позива

Мемоизација

Иако коректна, горња функција је неефикасна и може се поправити техником динамичког програмирања. Најједноставније прилагођавање је да се употреби мемоизација. Пошто функција има два параметра, за мемоизацију ћемо употребити матрицу. Ако се $\binom{n}{k}$ памти у матрици на позицији (n, k), матрицу можемо алоцирати на $n + 1$ врста, где последња врста има $n + 1$ елемената, а свака претходна један елемент мање (у матрицу ће се попуњавати елементи Паскаловог троугла). Пошто нас неће занимати вредности веће од полазног k и пошто се и k и n смањују током рекурзије, при чему је $k \leq n$, можемо и одсећи део троугла десно од позиције k.

Пошто су бројеви комбинација увек већи од нуле, вредности 0 у матрици ће нам означавати да позив за те параметре још није извршен.

Анализа сложености. И меморијска и временска сложеност мемоизоване верзије је $O(nk)$. Наиме, у дрвету рекурзивних позива се сваки чвор може јавити највише два пута. Пошто сваки чвор садржи пар бројева таквих да је први од 0 до k, а други од 0 до n, укупан број чворова је одозго ограничен са $2nk$ (а може бити и мањи, јер се неки парови не могу јавити).



Слика 8.6: Дрво рекурзивних позива уз мемоизацију

```
long long brojKombinacija(int k, int n, vector<vector<long long>>& memo) {
  // ako smo već računali broj kombinacija, ne računamo ga ponovo
  if (memo[n][k] != 0) return memo[n][k];

  // broj kombinacija na početku i na kraju svake vrste jednak je 1
  if (k == 0 || k == n) return memo[n][k] = 1;
  // broj kombinacija u sredini jednak je zbiru
  // broja kombinacija iznad i iznad levo od tekućeg elementa
  return memo[n][k] = brojKombinacija(k-1, n-1, memo) +
                      brojKombinacija(k, n-1, memo);
}

long long brojKombinacija(int K, int N) {
  // alociramo prostor za rezultate rekurzivnih poziva koji se
  // mogu desiti i popunjavamo matricu nulama
  vector<vector<long long>> memo(N+1);
  for (int n = 0; n <= N; n++)
    memo[n].resize(min(K+1, n+1), 0);
  // pozivamo funkciju koja će izračunati traženi broj
  return brojKombinacija(K, N, memo);
}
```

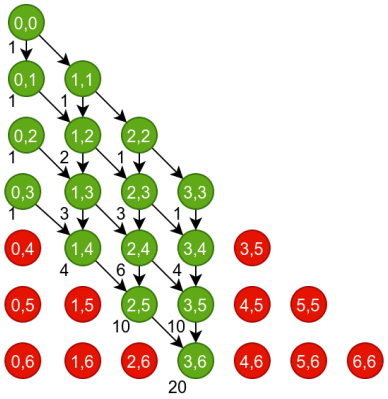
Динамичко програмирање навише

Уместо мемоизације можемо употребити и динамичко програмирање навише, ослободити се рекурзије и попунити троугао врсту по врсту наниже. Попуњавање целог троугла је прилично једноставно.

```
long long brojKombinacija(int K, int N) {
  // alociramo prostor za smeštanje celog trougla
  vector<vector<long long>> dp(N+1);
```

```
  for (int n = 0; n <= N; n++)
    dp[n].resize(n+1);
  // obrađujemo vrstu po vrstu
  for (int n = 0; n <= N; n++) {
    // na početku svake vrste nalazi se 1
    dp[n][0] = 1;
    // unutrašnje elemente izračunavamo kao zbir elemenata iznad njih
    for (int k = 1; k < n; k++)
      dp[n][k] = dp[n-1][k-1] + dp[n-1][k];
    // na kraju svake vrste nalazi se 1
    dp[n][n] = 1;
  }
  // vraćamo traženi rezultat
  return dp[N][K];
}
```

И у овом случају можемо одсећи непотребне десне колоне у троуглу. Могуће је одсећи и део испод одговарајуће дијагонале – таква одсецања се обично не ради у динамичком програмирању навише јер не мењају асимптотску сложеност, док их рекурзивна имплементација заснована на мемоизацији природно избегава. На наредној слици су обележени елементи Паскаловог троугла који су потребни за израчунавање броја $\binom{6}{3} = 20$.



Слика 8.7: Елементи Паскаловог троугла потребни за израчунавање броја $\binom{6}{3}$

```
long long brojKombinacija(int K, int N) {
  // alociramo prostor za smeštanje relevantnog dela trougla
  vector<vector<long long>> dp(N+1);
  for (int n = 0; n <= N; n++)
    dp[n].resize(min(K+1, n+1));
  // trougao popunjavamo kolonu po kolonu
  for (int n = 0; n <= N; n++) {
    // na početku svake vrste nalazi se 1
    dp[n][0] = 1;
    // unutrašnje elemente izračunavamo kao zbir elemenata iznad njih
    for (int k = 1; k <= min(n-1, K); k++)
      dp[n][k] = dp[n-1][k-1] + dp[n-1][k];
    // ako je potrebno da znamo krajnji element kolone, postavljamo
    // ga na vrednost 1
    if (n <= K)
       dp[n][n] = 1;
  }
  // vraćamo traženi rezultat
  return dp[N][K];
}
```

Пажљивијом анализом претходног кода видимо да, како је то обично случај у динамичком програмирању, не морамо истовремено чувати све елементе матрице, јер свака врста зависи само од претходне и довољно

је уместо матрице чувати само њене две врсте (претходну и текућу). Заправо, довољно је чувати само један вектор врсту ако је пажљиво попуњавамо и ако током њеног ажурирања у једном њеном делу чувамо текућу, а у другом наредну врсту. Пошто елемент (n, k) зависи од елемента $(n-1, k-1)$ и од елемента $(n-1, k)$ значи да сваки елемент зависи од елемената који су лево од њега, али не од елемената који су десно од њега. Зато ћемо вектор попуњавати здесна налево. Претпоставићемо да током ажурирања важи инваријанта да се на позицијама строго већим од k налазе елементи врсте n, а да се на позицијама мањим или једнаким од k налазе елементи врсте $n-1$. Ажурирање започиње тиме што на крај врсте допишемо вредност 1 (осим у случају када вршимо сасецање десног дела троугла) и наставља се тако што се елемент на позицији k увећа за вредност на позицији $k-1$. Заиста, пре ажурирања се на позицији k налази вредост троугла са позиције $(n-1, k)$, док се на позицији $k-1$ налази вредност троугла са позиције $(n-1, k-1)$. Њихов збир је вредност троугла на позицији (n, k), па се он уписује на позицију k и након тога се k смањује за 1, чиме се инваријанта одржава. Ажурирање се врши до позиције $k = 1$, јер се на позицији $k = 0$ у свим врстама налази вредност 1.

Меморијска сложеност овог решења је $O(k)$, док је временска $O(n \cdot k)$. Приметимо како смо од веома неефикасног решења експоненцијалне сложености техником динамичког програмирања добили веома ефикасно и уз то прилично једноставно решење.

```
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 vrstu po vrstu
  for (int n = 1; n <= N; n++) {
    // vrstu ažuriramo 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];
}
```

Другачија рекурзивна дефиниција

Рецимо и да смо могли кренути од алгоритма набрајања свих комбинација у ком се у петљи разматрају сви кандидати за елемент на текућој позицији. Тиме би се добио алгоритам који би елемент $\binom{n}{k}$ рачунао по следећој формули:

$$\binom{n}{k} = \sum_{n'=k}^{n} \binom{n'}{k-1}.$$

Задатак: Дигитални бројач

$2n$-то цифрени дигитални бројач који одбројава од $000\ldots000$ до $999\ldots999$ емитује звучни сигнал сваки пут када је сума првих n цифара једнака суми последњих n цифара. На пример, за шестоцифрени дигитални бројач звучни сигнал се пушта за $000000, 001001, 001010, \ldots, 999999$. Написати програм који ће одредити колико пута ће бити емитован звучни сигнал.

Улаз: У првој линији стандардног улаза налази се природан број n ($1 \leq n \leq 9$).

Излаз: На стандарном излазу приказати колико постоји $2n$-цифрених бројева са траженим својством.

Пример

Улаз	*Излаз*
3	55252

Решење

Наивна решења

Директно решење је да се у петљи прођу сви бројеви од $00\ldots0$ до $99\ldots9$, да се за сваки број одреди збир леве и десне половине, и ако су они једнаки, да се увећа бројач.

Анализа сложености. Пошто постоји 10^{2n} бројева са $2n$ цифара и пошто се за сваки од њих збир цифара сваке половине израчунава у n корака, сложеност овог алгоритма је $O(2n \cdot 10^{2n})$.

```
// uklanja poslednjih n cifara broja x odredjujuci njihov zbir
int zbirNPoslednjihCifara(long long& x, int n) {
  int zbir = 0;
  for (int i = 0; i < n; i++) {
    zbir += x % 10;
    x /= 10;
  }
  return zbir;
}

// odredjuje zbirove prve i druge polovine broja x koji ima 2n cifara
void zbiroviCifara(long long x, int n,
                   int& zbirPrvePolovine, int& zbirDrugePolovine) {
  zbirDrugePolovine = zbirNPoslednjihCifara(x, n);
  zbirPrvePolovine = zbirNPoslednjihCifara(x, n);
}

// izracunava broj 2n-tocifrenih brojeva kod kojih je zbir prvih n i
// zbir drugih n cifara jednak
int brojBrojeva(int n) {
  long long ukupnoBrojeva = 0;
  long long max = (long long)pow(10, 2*n) - 1;
  for (long long i = 0; i <= max; i++) {
    int zbirPrvePolovine, zbirDrugePolovine;
    zbiroviCifara(i, n, zbirPrvePolovine, zbirDrugePolovine);
    if (zbirPrvePolovine == zbirDrugePolovine)
      ukupnoBrojeva++;
  }
  return ukupnoBrojeva;
}
```

Једно решење је да се лева и десна половина броја набрајају у две угнежђене петље и броји колико пута ће збир цифара спољашње циклусне променљиве бити једнак збиру цифара унутрашње циклусне променљиве. Збир леве половине броја се тада може рачунати само једном по извршавању целокупне унутрашње петље.

Анализа сложености. Сложеност овог алгоритма је $O(n \cdot 10^{2n})$ (кључни проблем је то што се сваки од 10^n n-тоцифрених бројева комбинује са сваким од 10^n n-тоцифрених бројева, што захтева 10^{2n} корака).

```
// izracunava zbir cifara u dekadnom zapisu broja x
int zbirCifara(long long x) {
  int zbir = 0;
  while (x > 0) {
    zbir += x % 10;
    x /= 10;
  }
  return zbir;
}

// izracunava broj 2n-tocifrenih brojeva kod kojih je zbir prvih n i
// zbir drugih n cifara jednak
int brojBrojeva(int n) {
  long long ukupnoBrojeva = 0;
  long long max = (long long)pow(10, n) - 1;
```

```
  for (int prvaPolovina = 0; prvaPolovina <= max; prvaPolovina++) {
    int zbirPrvePolovine = zbirCifara(prvaPolovina);
    for (int drugaPolovina = 0; drugaPolovina <= max; drugaPolovina++) {
      int zbirDrugePolovine = zbirCifara(drugaPolovina);
      if (zbirPrvePolovine == zbirDrugePolovine)
        ukupnoBrojeva++;
    }
  }
  return ukupnoBrojeva;
}
```

Број n-тоцифрених бројева датог збира

Пошто лева и десна половина треба да имају исти збир (на пример, s), и једна и друга половина ће бити n-тоцифрени бројеви чији је збир цифара s. Збир цифара n-тоцифреног броја узима вредности од 0 (за број $00\ldots0$) до $9 \cdot n$ (за број $99\ldots9$). За свако такво s треба да одредимо на колико разних начина можемо да одаберемо два броја чији је збир цифара s. Означимо са b_s број n-тоцифрених бројева којима је збир цифара s (број између 0 и $9 \cdot n$). Дакле и прву и другу половину можемо изабрати на b_s начина (пошто оне не морају бити различите), тако да постоји укупно b_s^2 бројева чија лева и десна половина имају збир s. Зато ће тражени број бити једнак $b_0^2 + b_1^2 + ... + b_{9n}^2$.

Остаје питање како одредити број n-тоцифрених бројева чији је збир s.

Бројање бројева датог збира

Један начин је да се уведе низ бројача, по један за сваку вредност c. У петљи се обилазе сви бројеви од 0 до $10^n - 1$, за сваки се одређује збир цифара и увећава се бројач бројева добијеног збира.

Анализа сложености. Пошто се за сваки од 10^n n-тоцифрених бројева збир цифара израчунава у сложености $O(n)$, и пошто се коначни број рачуна у $9n$ корака петље тј. у сложености $O(n)$, сложеност овог алгоритма је $n \cdot 10^n$.

```
// izracunava broj 2n-tocifrenih brojeva kod kojih je zbir prvih n i
// zbir drugih n cifara jednak
long long brojBrojeva(int n) {
  // broj brojeva sa datim zbirom cifara
  vector<long long> brojBrojevaDatogZbiraCifara(9*n + 1, 0);

  // za sve n-tocifrene brojeve izmedju 00..0 i 99...9
  long long st10 = Stepen(10, n);
  for (long long broj = 0; broj < st10; broj++)
    // odredjujemo zbir cifara i uvecavamo broj brojeva sa tim zbirom
    // cifara
    brojBrojevaDatogZbiraCifara[ZbirCifara(broj)]++;

  // ukupan broj brojeva cija leva i desna polovina imaju isti broj cifara
  long long ukupnoBrojeva = 0;
  // za svaki moguci zbir cifara polovine broja
  for (int zbir = 0; zbir <= 9*n; zbir++) {
    // postoji b n-tocifrenih brojeva koji imaju zbir cifara zbir
    long long b = brojBrojevaDatogZbiraCifara[zbir];
    // levu polovinu broja mozemo odabrati na b nacina i desnu
    // polovinu na b nacina, tako da ukupno takvih brojeva ima b*b
    ukupnoBrojeva += b * b;
  }
  return ukupnoBrojeva;
}
```

Бројање партиција динамичким програмирањем

Један начин да се одреди број n-тоцифрених бројева чији је збир цифра b_s је да се примени рекурзивно решење по броју цифара.

Уведимо стога ознаку $b_{k,s}$ за број k-тоцифрених бројева чији је збир цифара s.

- Ако је $k = 1$, бројева $b_{1,s}$ има 1 за $0 \leq s \leq 9$, тј. 0 у супротном.
- Ако је $k > 1$ тада на прво место можемо поставити било коју цифру c од 0 до 9 (тј. до s ако је $s < 9$). Када њу поставимо, остале цифре можемо поставити на $b_{k-1,s-c}$ начина. Дакле,

$$b_{k,s} = \sum_{c=0}^{\min(s,9)} b_{k-1,s-c}.$$

Пошто се у овој рекрузивној формулацији рекурзивни позиви преклапају, потребно је употребити технику динамичког програмирања да би се решење убрзало. Једна могућност је да употребимо мемоизацију.

```
// broj nacina da se napravi broj koji ima brojCifara cifara i kome je
// zbir cifara jednak zbirCifara
long long brojParticija(int zbirCifara, int brojCifara) {
  // koristimo memoizaciju da bismo izbegli da vise puta racunamo
  // jedan te isti rezultat - najjednostavnija (ali ne i
  // najefikasnija) implementacija cuva ranije rezultate u recniku
  static map<pair<int, int>, long long> memo;
  // proveravamo da li smo vec racunali ovaj rezultat i ako jesmo
  // vracamo raniji rezultat
  auto p = make_pair(zbirCifara, brojCifara);
  if (memo.find(p) != memo.end())
    return memo[p];

  if (brojCifara == 1)
    // brojevi izmedju 0 i 9 se mogu na jedan nacin predstaviti kao
    // zbir jedne cifre, a veci brojevi od toga se ne mogu predstaviti
    return memo[p] = zbirCifara < 10 ? 1 : 0;
  else {
    long long rezultat = 0;
    // na prvo mesto postavljamo redom jednu po jednu cifru
    for (int cifra = 0; cifra <= 9 && cifra <= zbirCifara; cifra++)
      // rekurzivno racunamo broj particija preostalog zbira sa jednom
      // cifrom manje
      rezultat += brojParticija(zbirCifara - cifra, brojCifara - 1);
    return memo[p] = rezultat;
  }
}


// izracunava broj 2n-tocifrenih brojeva kod kojih je zbir prvih n i
// zbir drugih n cifara jednak
long long brojBrojeva(int n) {
  long long ukupnoBrojeva = 0;
  for (int zbirCifara = 0; zbirCifara <= 9*n; zbirCifara++) {
    long long b = brojParticija(zbirCifara, n);
    ukupnoBrojeva += b * b;
  }
  return ukupnoBrojeva;
}
```

Динамичко програмирање навише

Уместо мемоизације можемо употребити и динамичко програмирање навише. Вредности $b_{k,s}$ можемо чувати у матрици, чије попуњавање почињемо од првог реда (за $k = 1$) и попуњавамо је затим ред по ред, за растуће вредности k.

Пример. На пример, да бисмо одредили број $b_{3,15}$ попуњавамо следећу матрицу.

```
    0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
-------------------------------------------------
1 | 1  1  1  1  1  1  1  1  1  1   0  0  0  0  0  0
2 | 1  2  3  4  5  6  7  8  9 10   9  8  7  6  5  4
3 | 1  3  6 10 15 21 28 36 45 55  63 69 73 75 75 73
```

```
// alociramo matricu dimenzije m x n
vector<vector<long long>> alociraj(int m, int n) {
  vector<vector<long long>> rezultat(m);
  for (int i = 0; i < m; i++)
    rezultat[i].resize(n);
  return rezultat;
}

// vraca matricu koja za svaki brojCifara od 1 do maxBrojCifara i
// svaki zbirCifara od 1 do maxZbirCifara na mestu
// [brojCifara][zbirCifara] sadrzi broj brojeva koji imaju brojCifara
// cifara i zbir cifara jednak broju zbirCifara
vector<vector<long long>> brojParticija(int maxBrojCifara, int maxZbirCifara) {
  vector<vector<long long>> dp = alociraj(maxBrojCifara + 1, maxZbirCifara + 1);

  // brojevi izmedju 0 i 9 se mogu na jedan nacin predstaviti kao zbir
  // jedne cifre, a veci brojevi od toga se ne mogu predstaviti
  for (int zbirCifara = 0; zbirCifara <= maxZbirCifara; zbirCifara++)
    dp[1][zbirCifara] = zbirCifara < 10 ? 1 : 0;

  // odredjujemo broj razlaganja na vise cifara
  for (int brojCifara = 2; brojCifara <= maxBrojCifara; brojCifara++) {
    // zbir moramo da obilazimo unazad da bismo mogli da koristimo
    // samo jedan niz
    for (int zbirCifara = 0; zbirCifara <= maxZbirCifara; zbirCifara++) {
      dp[brojCifara][zbirCifara] = 0;
      // na prvo mesto postavljamo redom jednu po jednu cifru
      for (int cifra = 0; cifra <= 9 && cifra <= zbirCifara; cifra++)
        // rekurzivno racunamo broj particija preostalog zbira sa jednom
        // cifrom manje
        dp[brojCifara][zbirCifara] += dp[brojCifara - 1][zbirCifara - cifra];
    }
  }
  return dp;
}

// izracunava broj 2n-tocifrenih brojeva kod kojih je zbir prvih n i
// zbir drugih n cifara jednak
long long brojBrojeva(int n) {
  // za svaki broj od 0 do 9*n nas zanima koliko postoji razlicitih
  // n-tocifrenih brojeva ciji je zbir cifara taj broj
  vector<vector<long long>> dp = brojParticija(n, 9*n);

  // ukupan broj brojeva cija leva i desna polovina imaju isti broj cifara
  long long ukupnoBrojeva = 0;
  // za svaki moguci zbir cifara polovine broja
  for (int zbirCifara = 0; zbirCifara <= 9*n; zbirCifara++)
    // postoji dp[n][zbirCifara] nacina da napravimo levu polovinu i
    // dp[n][zbirCifara] nacina da napravimo desnu polovinu broja
    // tj. dp[zbirCifara]*dp[zbirCifara] nacina da odaberemo broj kome
    // i leva i desna polovina imaju zbir cifara zbirCifara
    ukupnoBrojeva += dp[n][zbirCifara] * dp[n][zbirCifara];
```

```
  return ukupnoBrojeva;
}
```

Анализа сложености. Максимални број цифара је n, а максимални збир цифара је $9n$, па је димензија матрице $(n+1) \times (9n+1)$. Дакле, попуњава се $O(n^2)$ поља матрице. Свако поље захтева највише 10 корака сабирања (за цифре од 0 до 9), па се свако поље попуњава у времену (1), додуше уз велики константни фактор. То би се могло поправити ако би се чували парцијални збирови елемената сваке врсте. Употреба парцијалних збирова за оптимизацију израчунавања збирова сегмената датог низа приказана је, на пример, у задатку Збирови сегмената. Након попуњавања матрице, број елемената се израчунава у времену $O(n)$, без коришћења додатне меморије. Дакле, и временска и меморијска сложеност је $O(n^2)$.

Меморијска оптимизација

Приметимо да се за израчунавање вредности у реду k користе само вредности у реду $k-1$. Зато није потребно чувати целу матрицу, већ само њену претходну врсту. Додатно, ако приметимо да за израчунавање вредности $b_{k,s}$ нису потребне вредности из претходног реда за збирове $s' > s$, довољно је да податке чувамо само у једном низу у којем ћемо вредности наредног реда од вредности претходног добијати тако што ћемо кренути од краја и ажурирати једну по једну вредност. Пошто $b_{k,s}$ зависи од $b_{k-1,s}$, потребна нам је једна помоћна променљива у којој ћемо израчунати резултат пре него што га упишемо на место s у низу или вредност можемо добити тако што вредност на месту s која одговара цифри 0 увећамо за вредности на позицијама $s-c$ које одговарају цифрама 1, 2 и тако даље.

Анализа сложености. Димензија матрице је $O(n^2)$, што је и време потребно за њено попуњавање (уз велики константни фактор, због сабирања 10 елемената). Укупна временска сложеност овог приступа је $O(n^2)$, јер је након попуњавања матрице потребно још само $O(n)$ операција. Пошто није потребно чувати целу матрицу, већ само један текући ред, меморијска сложеност је само $O(n)$.

```
// za svaki broj iz [0, maxZbir] odredjujemo broj nacina da se taj
// broj predstavi kao zbir brojCifara cifara (pri cemu se u obzir
// uzima i redosled cifara)
vector<long long> brojParticija(int brojCifara, int maxZbir) {
  // resenje gradimo dinamickim programiranje
  vector<long long> dp(maxZbir + 1);

  // brojevi izmedju 0 i 9 se mogu na jedan nacin predstaviti kao zbir
  // jedne cifre, a veci brojevi od toga se ne mogu predstaviti
  for (int zbir = 0; zbir <= maxZbir; zbir++)
    dp[zbir] = zbir < 10 ? 1 : 0;

  // odredjujemo broj razlaganja na vise cifara
  for (int cifara = 2; cifara <= brojCifara; cifara++) {
    // zbir moramo da obilazimo unazad da bismo mogli da koristimo
    // samo jedan niz
    for (int zbirCifara = maxZbir; zbirCifara >= 0; zbirCifara--) {
      // na prvo mesto postavljamo redom jednu po jednu cifru
      for (int cifra = 1; cifra <= 9 && cifra <= zbirCifara; cifra++)
        // rekurzivno racunamo broj particija preostalog zbira sa jednom
        // cifrom manje
        dp[zbirCifara] += dp[zbirCifara - cifra];
    }
  }
  return dp;
}

// izracunava broj 2n-tocifrenih brojeva kod kojih je zbir prvih n i
// zbir drugih n cifara jednak
long long brojBrojeva(int n) {
  // za svaki broj od 0 do 9*n nas zanima koliko postoji razlicitih
  // n-tocifrenih brojeva ciji je zbir cifara taj broj
  vector<long long> bp = brojParticija(n, 9*n);
```

```
  // ukupan broj brojeva cija leva i desna polovina imaju isti broj cifara
  long long ukupnoBrojeva = 0;
  // za svaki moguci zbir cifara polovine broja
  for (int zbirCifara = 0; zbirCifara <= 9*n; zbirCifara++) {
    // postoji bp[zbirCifara] nacina da napravimo levu polovinu i
    // bp[zbirCifara] nacina da napravimo desnu polovinu broja
    // tj. bp[zbirCifara]*bp[zbirCifara] nacina da odaberemo broj kome
    // i leva i desna polovina imaju zbir cifara zbirCifara
    ukupnoBrojeva += bp[zbirCifara] * bp[zbirCifara];
  }
  return ukupnoBrojeva;
}
```

Задатак: Број појављивања подниске

Написати програм којим се за две дате ниске x и y, одређује број појављивања ниске y као подниза ниске x. Ниска y је подниз ниске x ако се може добити од ниске x брисањем произвољног броја карактера.

Улаз: Прва линија стандардног улаза садржи ниску x, а друга линија ниску y. Дужине ниски су највише 100 карактера.

Излаз: На стандардном изласу приказати само број појављивања ниске y као подниза ниске x.

Пример

Улаз	*Излаз*
`abcbca`	`3`
`abc`	

Објашњење:

Позиције појављивања подниза су обележене великим словима.

```
ABCbca
ABcbCa
AbcBCa
```

Решење

Размотримо прво рекурзивно решење.

- Ако су обе ниске непразне, тада се може упоредити њихово последње слово. Ако су им последња слова различита онда је укупан број појављивања подниске једнак броју њених појављивања у префиксу прве ниске без последњег карактера, а ако су им последња слова једнака, онда је укупан број појављивања једнак том броју увећаном за број појављивања префикса друге ниске без последњег карактера у префиксу прве ниске без последњег карактера.
- Базу чине случајеви када је нека ниска празна. Ако је друга ниска празна тада се онда једном јавља као подниз прве ниске, а ако је прва ниска празна (а друга није) тада она не садржи подниз који одговара другој ниски.

Пошто се кроз рекурзију стално разматрају префикси ниски, ниске не морамо мењати кроз рекурзију, већ можемо прослеђивати само дужине префикса (нека је n дужина префикса прве, а m дужина префикса друге ниске). Ако је $f(n, m)$ тражени број појављивања, тада важи:

$$
\begin{aligned}
f(n, 0) &= 1 \\
f(0, m) &= 0, \quad \text{за } m > 0 \\
f(n, m) &= f(n-1, m) + f(n-1, m-1), \quad \text{за } n, m > 0 \ \ a_n = a_m \\
f(n, m) &= f(n-1, m), \quad \text{за } n, m > 0 \ \ a_n \neq a_m
\end{aligned}
$$

На основу овога је веома једноставно направити рекурзивну имплементацију.

```
long long brojPojavljivanjaPodniske(const string& niska, int duzina_niske,
                                    const string& podniska, int duzina_podniske) {
  if (duzina_podniske == 0)
    return 1;
  if (duzina_niske == 0)
    return 0;
  long long broj = brojPojavljivanjaPodniske(niska, duzina_niske-1,
                                             podniska, duzina_podniske);
  if (niska[duzina_niske-1] == podniska[duzina_podniske-1])
    broj += brojPojavljivanjaPodniske(niska, duzina_niske-1,
                                      podniska, duzina_podniske-1);
  return broj;
}

long long brojPojavljivanjaPodniske(const string& niska,
                                    const string& podniska) {
  return brojPojavljivanjaPodniske(niska, niska.length(), podniska, podniska.length());
}
```

С обзиром на то да ће се у наивној рекурзивној имплементацији исти рекурзивни позиви извршавати више пута, таква имплементација биће неефикасна. Ефикасност се лако може поправити техником динамичког програмирања (било мемоизацијом, било динамичким програмирањем навише). Пошто функција има два параметра, вредности рекурзивних позива можемо памтити у матрици.

Пример. Прикажимо ову матрицу на примеру бројања појављивања подниске `abc` унутар ниске `abcbca`.

```
    0  1  2  3
  | -----------
0 | 1  0  0  0
1 | 1  1  0  0
2 | 1  1  1  0
3 | 1  1  1  1
4 | 1  1  2  1
5 | 1  1  2  3
6 | 1  2  2  3
```

Пошто елемент на позицији (n, m) зависи од елемената на позицијама $(n-1, m)$ и $(n-1, m-1)$ матрицу можемо попуњавати било врсту по врсту, било колону по колону.

```
long long brojPojavljivanjaPodniskaa(const string& niska,
                                     const string& podniska) {
  int n = niska.length();
  int m = podniska.length();
  vector<vector<long long>> dp(n + 1);
  for (int i = 0; i <= n; i++)
    dp[i].resize(m + 1);

  for (int i = 0; i <= n; i++)
    dp[i][0] = 1;
  for (int j = 1; j <= m; j++)
    dp[0][j] = 0;
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= m; j++) {
      dp[i][j] = dp[i-1][j];
      if (niska[i-1] == podniska[j-1])
        dp[i][j] += dp[i-1][j-1];
    }
  return dp[n][m];
}
```

Ако претпоставимо да попуњавамо врсту по врсту матрице можемо извршити меморијску оптимизацију тако

што само чувамо елементе једне врсте. У првој врсти иницијализујемо све елементе на нулу, осим првог који иницијализујемо на јединицу. Приликом преласка са текуће на наредну врсту, ажурирање можемо вршити с десна на лево. Ако су одговарајући карактери ниски једнаки, тада текући елемент увећавамо за њему претходни.

```
long long brojPojavljivanjaPodniske(const string& niska,
                                    const string& podniska) {
  int n = niska.length();
  int m = podniska.length();
  vector<long long> dp(m + 1, 0);
  dp[0] = 1;
  for (int i = 1; i <= n; i++)
    for (int j = m; j >= 1; j--)
      if (niska[i-1] == podniska[j-1])
        dp[j] += dp[j-1];
  return dp[m];
}
```

8.3 Оптимизација коришћењем динамичког програмирања

Динамичко програмирање се често примењује у решавању оптимизационих задатака (задатака у којима се тражи да се одреди најмања или највећа вредност која задовољава неки услов). Да би оптимизациони задатак могао да се решава динамичким програмирањем, потребно је формулисати његово рекурзивно решење, што значи да проблем мора да задовољава такозвано својство **оптималне подструктуре**. То значи да се оптимално решење полазног проблема може одредити на основу оптималних решења потпроблема истог облика, али мање димензије.

На пример, најкраћи пут између тачке A и тачке B који пролази преко тачке C се добија тако што се искомбинују најкраћи пут између тачке A и тачке C и накраћи пут између тачке C и тачке B, што значи да проблем одређивања најкраћих путева има својство оптималне подструктуре и може се решавати динамичким програмирањем (на тој техници је заснован Дајкстрин алгоритам, који је један од најчувенијих алгоритама за решвање тог проблема).

Међутим, најјефтинији авионски лет од аеродрома A до аеродрома B преко аеродрома C не мора да буде комбинација најјефтинијих летова од A до C и од C до B (због посебних попуста које авио-компаније нуде за повезане летове), тако да се проблем одређивања најјефтинијег лета нема својство оптималне подструктуре и не може се решавати динамичким програмирањем.

Задатак: Максимални збир на путу кроз матрицу

У табели димензија $n \times n$ поља су попуњена цифрама од 0 до 9. Играч који се налази у горњем левом углу табеле може да у једном кораку пређе у суседно десно поље или суседно доње поље. Циљ му је да стигне до доњег десног поља тако да збир вредности на пређеним пољима буде максималан. Написати програм којим се одређује максимални збир коју може остварити играч при кретању од горњег левог до доњег десног угла.

Улаз: У првој линији стандардног улаза се уноси број редова табеле n ($1 \leq n \leq 30$), а у следећих n редова по n цифара од 0 до 9.

Излаз: У првој линији стандардног излаза приказати тражену вредност максималног збира.

Пример

Улаз	*Излаз*
5 4 3 5 7 5 1 9 4 1 3 2 3 5 1 2 1 3 1 2 0 4 6 7 2 1	38

Решење

Рекурзивно решење

Покушајмо прво да дефинишемо рекурзивно решење. Дефинишемо рекурзивну функцију која одређује максимални збир вредности који се може постићи крећући се од поља $(0,0)$ до датог поља (v,k), које је параметар функције. Тражени резултат добијамо када је (v,k) једнако $(n-1,n-1)$.

Излаз из рекурзије представља случај када је $(v,k)=(0,0)$ – постоји само једна путања од поља $(0,0)$ до њега самог (када се не померамо) и збир вредности на њој је једнак вредности на пољу $(0,0)$.

У супротном посматрајмо неко поље (v,k) различито од $(0,0)$. На то смо поље могли доћи или са поља $(v-1,k)$ или са поља $(v,k-1)$, при чему први случај није могућ када се поље налази у почетној врсти (када је $v=0$), а други није могућ када се поље налази у почетној колони (када је $k=0$). Да би се постигао оптимални збир до поља (v,k), мора се постићи оптимални збир до поља са ког смо на њега прешли. Заиста ако би се до претходног поља могло доћи путањом са већим збиром, тада бисмо ту путању могли проширити последњим преласком на поље (v,k) и добити већу вредност збира на путањи до поља (v,k). Зато вредност одређујемо додајући елемент матрице на позицији (v,k) на вредности рекурзивних позива за поља $(v-1,k)$ и $(v,k-1)$ (водивши рачуна да се прескочи неодговарајући рекурзивни позив у случајевима када је $v=0$ тј. $k=0$) и бирајући повољнију од те две варијанте (ону која даје већи збир).

Ако са $z(v,k)$ обележимо вредност максималног збира до поља (v,k), тада важи следеће:

$$z(v,k)=\begin{cases}0 & \text{ако је } v=0 \text{ и } k=0\\ z(v,k-1) & \text{ако је } v=0 \text{ и } k>0\\ z(v-1,k) & \text{ако је } v>0 \text{ и } k=0\\ \max(z(v-1,k),z(v,k-1)) & \text{ако је } v>0 \text{ и } k>0\end{cases}$$

Овај проблем поседује својство оптималне подструктуре. Рекурзивно решење овог оптимизационог проблема је било могуће захваљујући томе што овај проблем задовољава принцип оптималности, по коме оптимално решење проблема мора садржати и оптимално решење сваког свог потпроблема. У овом проблему то својство је испуњено јер оптимална путања, која представља решење проблема, мора бити оптимална на сваком делу пута.

Пример. Ради илустровања проблема размотримо квадратну матрицу A следећег садржаја:

```
4   3   5   7   5
1   9   4   1   3
2   3   5   1   2
1   3   1   2   0
4   6   7   2   1
```

Вредност највећег збира до сваког од поља може се израчунати на основу претходне рекурзивне дефиниције и може се представити следећом матрицом.

```
4   7 12 19 24
5  16 20 21 27
7  19 25 26 29
8  22 26 28 29
12 28 35 37 38
```

```
int maksZbir(const vector<vector<int>>& M, int n, int v, int k) {
    if (v == 0 && k == 0)
        return M[v][k];
    int odGore = 0, sLeva = 0;
    if (v > 0)
       odGore = M[v][k] + maksZbir(M, n, v-1, k);
    if (k > 0)
       sLeva = M[v][k] + maksZbir(M, n, v, k-1);
    if (v == 0) return sLeva;
    if (k == 0) return odGore;
    return max(odGore, sLeva);
}
```

```
int maksZbir(const vector<vector<int>>& M) {
  int n = M.size();
  return maksZbir(M, n, n-1, n-1);
}
```

Анализа сложености. У рекурзивној имплементацији долази до понављања истих рекурзивних позива (на пример, ако направимо кораке `dole-desno` и `desno-dole`, наћи ћемо се на истом пољу). Стога је претходна имплементација изразито неефикасна (сложеност најгорег случаја је експоненцијална).

Могуће је дати и дуалну рекурзивну дефиницију, којом се израчунава оптимални пут од датог поља (v, k) до крајњег поља $(n-1, n-1)$. Излаз из рекурзије је када је $(v, k) = (n-1, n-1)$, док у рекурзивним корацима анализирамо два поља до којих можемо доћи са текућег (то су оно десно и доле од њега) и за њих рекурзивно позивамо функцију.

```
int maksZbir(const vector<vector<int>>& M, int n, int v, int k) {
  if (v >= n-1 && k >= n-1)
    return M[v][k];
  int dole, desno;
  if (v < n - 1)
    dole = M[v][k] + maksZbir(M, n, v+1, k);
  if (k < n - 1)
    desno = M[v][k] + maksZbir(M, n, v, k+1);
  if (v >= n-1) return desno;
  if (k >= n-1) return dole;
  return max(dole, desno);
}

int maksZbir(const vector<vector<int>>& M) {
  return maksZbir(M, M.size(), 0, 0);
}
```

Мемоизација

Решење директном рекурзијом је неефикасно и потребно је извршити оптимизацију техником динамичког програмирања. Рекурзивну функцију је могуће проширити коришћењем мемоизације. Резултат за свако поље ћемо памтити у матрици.

У језику C++ матрицу можемо представити помоћу вектора који садржи векторе и који се динамички алоцира током рада програма, када је позната димензија учитане матрице. Ово решење је елегантно и флексибилно, али се мора напоменути да се одређено време губи на динамичку алокацију, тако да се мало бржи програм може добити ако би матрица била унапред статички алоцирана (по цену, већег губитка меморије у случајевима када је димензија учитане матрице мања од максималне димензије допуштене текстом задатка).

Анализа сложености. За сваки пар бројева (v, k), такав да је $0 \leq v, k < n$, израчунавање се врши само једном, а затим се израчуната вредност очитава из матрице. Меморијска и временска сложеност овог решења су стога $O(n^2)$.

```
int maksZbir(const vector<vector<int>>& M, int n, int v, int k,
             vector<vector<int>>& memo) {
  if (memo[v][k] != -1)
    return memo[v][k];

  if (v == 0 && k == 0)
    return memo[v][k] = M[v][k];
  int odGore = 0, sLeva = 0;
  if (v > 0)
    odGore = M[v][k] + maksZbir(M, n, v-1, k, memo);
  if (k > 0)
    sLeva = M[v][k] + maksZbir(M, n, v, k-1, memo);
  if (v == 0) return memo[v][k] = sLeva;
  if (k == 0) return memo[v][k] = odGore;
  return memo[v][k] = max(odGore, sLeva);
```

```
}

int maksZbir(const vector<vector<int>>& M) {
  int n = M.size();
  vector<vector<int>> memo(n, vector<int>(n, -1));
  return maksZbir(M, n, n-1, n-1, memo);
}
```

Динамичко програмирање навише

Уместо мемоизације можемо употребити и динамичко програмирање навише. Резултат рекурзивне функције која израчунава максималну вредност збира од поља $(0, 0)$ до поља (v, k), памтићемо у помоћној матрици на њеном пољу (v, k). Пошто елемент у матрици може да зависи од елемента изнад и лево од себе, матрицу можемо попуњавати врсту по врсту (а могуће би било попуњавати је и колону по колону). Вредност на пољу $(0, 0)$ преписаћемо из полазне матрице, остале вредности у првој врсти ћемо добити увећавањем вредности лево од њих одговарајућим елементом полазне матрице, а остале вредности у првој колони ћемо добити увећавањем вредности изнад њих одговарајућим елементом полазне матрице. Остале вредности у матрици ћемо добити проналажењем већег од елемента изнад и лево од њих и увећавањем тог већег броја за одговарајућу вредност у полазној матрици.

```
int maksZbir(const vector<vector<int>>& M) {
  int n = M.size();
  vector<vector<int>> dp(n);
  for (int i = 0; i < n; i++)
    dp[i].resize(n);
  dp[0][0] = M[0][0];
  for (int v = 1; v < n; v++)
    dp[v][0] = dp[v-1][0] + M[v][0];
  for (int k = 1; k < n; k++)
    dp[0][k] = dp[0][k-1] + M[0][k];
  for (int v = 1; v < n; v++)
    for (int k = 1; k < n; k++)
      dp[v][k] = max(dp[v-1][k] + M[v][k], dp[v][k-1] + M[v][k]);

  return dp[n-1][n-1];
}
```

Меморијска оптимизација

Можемо приметити да елементи у свакој врсти матрице када се користи динамичко програмирање навише зависе само од вредности у претходној врсти (а не од вредности у врстама пре ње). То нам омогућава да направимо додатну оптимизацију тако што ћемо уместо матрице за динамичко програмирање користити само један помоћни низ (у ком ћемо памтити елементе претходне врсте, а затим их један по један мењати елементима текуће врсте).

Анализа сложености. Временска сложеност овог решења остаје $O(n^2)$, али меморијска сложеност се смањује на $O(n)$.

```
int maksZbir(const vector<vector<int>>& M) {
  int n = M.size();
  vector<int> dp(n);
  dp[0] = M[0][0];
  for (int k = 1; k < n; k++)
    dp[k] = dp[k-1] + M[0][k];
  for (int v = 1; v < n; v++) {
    dp[0] += M[v][0];
    for (int k = 1; k < n; k++)
      dp[k] = max(dp[k] + M[v][k], dp[k-1] + M[v][k]);
  }
```

```
  return dp[n-1];
}
```

Задатак: Максимални пут кроз матрицу

У табели димензија $n \times n$ поља су попуњена цифрама од 0 до 9. Играч који се налази у горњем левом углу табеле може да у једном кораку пређе у суседно десно поље или суседно доње поље. Циљ му је да стигне до доњег десног поља тако да збир вредности на пређеним пољима буде максималан. Написати програм којим се одређује максимални збир коју може остварити играч при кретању од горњег левог до доњег десног угла и инструкције кретања: `desno`, `dole`, којима се обезбеђује кретање преко поља која дају максимални збир (ако има више могућих путева, исписати било који).

Улаз: У првој линији стандардног улаза се уноси број редова табеле n ($1 \leq n \leq 30$), а у следећих n редова по n цифара од 0 до 9.

Излаз: У првој линији стандардног излаза приказати тражену вредност максималног збира, а у наредним линијама исписати инструкције за кретање: `desno`, `dole` - у сваком реду по једну, којима се обезбеђује кретање преко поља која дају максимални збир.

Пример

Улаз	*Излаз*
5 4 3 5 7 5 1 9 4 1 3 2 3 5 1 2 1 3 1 2 0 4 6 7 2 1	38 desno dole dole dole dole desno desno desno

Решење

Овај задатак представља проширење задатка Максимални збир на путу кроз матрицу. Након одређивања оптималног пута, потребно је реконструисати и сам пут. Иако је понекад потребно складиштити и додатне информације да би се од вредности решења могло реконструисати решење, овде то није случај - решење је у потпуности могуће реконструисати на основу матрице изграђене у склопу динамичког програмирања навише (или у склопу мемоизације). Ипак, нагласимо да нам је потребна цела матрица и да није могуће извршити оптимизацију у којој се чува само текућа врста матрице.

Током реконструкције решења крећемо уназад, од завршног поља па све до почетног и на основу вредности у матрици закључујемо са ког претходног поља се стигло на текуће. Анализирамо поље лево и поље изнад текућег и гледамо на ком од њих пише већа вредност (ако су вредности једнаке, свеједно је које ћемо поље одабрати). Када знамо са ког смо поља стигли на текуће, знамо и на основу које инструкције робот прави тај корак (ако смо дошли са поља лево, инструкција је `desno`, а ако смо дошли са поља изнад, инструкција је `dole`). Потребно је само да посебно обратимо пажњу на поља у првој врсти и првој колони, јер код њих постоји само једна могућност. Крај реконструкције наступа када дођемо до поља $(0, 0)$. Приметимо да на овај начин, крећући се од завршног ка полазном пољу одређујемо низ инструкција унатраг, у обратном редоследу. Да бисмо добили инструкције у редоследу који је захтеван, можемо употребити рекурзивну функцију, тако што прво рекурзивно испишемо све инструкције за долазак на претходно поље и тек након тога испишемо инструкцију за прелазак са претходног на текуће поље.

```
// odredjujemo matricu koja na polju (v, k) sadrzi duzinu
// najkraceg puta od (0, 0) do (v, k)
vector<vector<int>> maksZbirDP(const vector<vector<int>>& M) {
  int n = M.size();
  vector<vector<int>> dp(n);
  for (int i = 0; i < n; i++)
    dp[i].resize(n);
  dp[0][0] = M[0][0];
  for (int v = 1; v < n; v++)
    dp[v][0] = dp[v-1][0] + M[v][0];
```

```
  for (int k = 1; k < n; k++)
    dp[0][k] = dp[0][k-1] + M[0][k];
  for (int v = 1; v < n; v++)
    for (int k = 1; k < n; k++)
      dp[v][k] = max(dp[v-1][k] + M[v][k], dp[v][k-1] + M[v][k]);
  return dp;
}

// ispisuje instrukcije kretanja od polja (0, 0) do polja (v, k)
void pisiPut(int v, int k, const vector<vector<int>>& dp) {
  // ako smo vec na polju (0, 0), nema potrebe da se pomeramo
  if (v == 0 && k == 0)
    return;

  if (v == 0) {
    // na polja u prvoj vrsti mozemo stici samo sa polja levo od njih,
    // pomerajuci se desno
    pisiPut(v, k-1, dp);
    cout << "desno" << endl;
  } else if (k == 0) {
    // na polja u prvoj koloni mozemo stici samo sa polja iznad njih,
    // pomerajuci se na dole
    pisiPut(v-1, k, dp);
    cout << "dole" << endl;
  } else
    // u suprotnom analiziramo da li nam je povoljnije da stignemo sa
    // polja iznad ili sa polja levo
    if (dp[v-1][k] > dp[v][k-1]) {
      pisiPut(v-1, k, dp);
      cout << "dole" << endl;
    } else {
      pisiPut(v, k-1, dp);
      cout << "desno" << endl;
    }
}
```

Уместо рекурзије можемо употребити и стек. Током пута уназад инструкције за прелазак са претходног на текуће поље ћемо постављати на помоћни стек, а затим, када стигнемо до поља $(0, 0)$, скидаћемо једну по једну вредност са стека и исписиваћемо је.

```
// ispisuje instrukcije kretanja
void pisiPut(const vector<vector<int>>& dp, int n) {
  // instrukcije dobijamo u obratnom redosledu, pa ih obrcemo
  // koriscenjem pomocnog steka
  stack<string> put;

  // krecemo od kraja i idemo ka pocetku
  int v = n-1, k = n-1;
  while (v > 0 || k > 0) {
    if (v == 0) {
      // na polja u prvoj vrsti mozemo stici samo sa polja levo od njih,
      // pomerajuci se desno
      k--;
      put.push("desno");
    } else if (k == 0) {
      // na polja u prvoj koloni mozemo stici samo sa polja iznad njih,
      // pomerajuci se na dole
      v--;
      put.push("dole");
```

```
    } else
      // u suprotnom analiziramo da li nam je povoljnije da stignemo sa
      // polja iznad ili sa polja levo
      if (dp[v-1][k] > dp[v][k-1]) {
        v--;
        put.push("dole");
      } else {
        k--;
        put.push("desno");
      }
  }

  // sadrzaj steka ispisujemo u obratnom redosledu
  while (!put.empty()) {
    cout << put.top() << endl;
    put.pop();
  }
}
```

Задатак: Исплата са најмање новчића

Дате су вредности n врста новчића. Написати програм који одређује минималан број новчића потребних за исплату датог износа S, при чему се може се користити и више новчића исте врсте и сваке врсте новчића има произвољно много. Вредности новчића и износ за исплату дати су у истој валути.

Улаз: Прва линија стандардног улаза задржи природан број S ($S \leq 1000$). Друга линија садржи природан број n ($n \leq 100$), у следећих n линија налазе се природни бројеви који представљају вредности за сваку врсту новчића, свака вредност у посебној линији.

Излаз: На стандардном излазу приказати у једној линији минималан број новчића потребан за исплату износа S.

Пример 1

Улаз	*Излаз*
7 3 1 3 2	3

Објашњење: исплата за износ 7 са најмање новчића је $3, 3, 1$.

Пример 2

Улаз

```
12
3
1
9
6
```

Излаз

```
2
```

Објашњење: исплата за износ 12 са најмање новчића је $6, 6$.

Решење

Анализа последње врсте новчића

Износ 0 се може наплатити са 0 новчића. У супротном, ако је низ расположивих новчића празан, износ није могуће наплатити. У супротном испитујемо могућност да је у износ укључен новчић последње врсте.

Ако није, покушавамо да наплатимо износ без коришћења последње врсте новчића, тј. са префиксном низа новчића без последњег елемента. Ако јесте, тада износ мора бити већи или једнак од новчића последње врсте и тада преостали износ покушавамо да наплатимо поново коришћењем свих врста новчића. Ако са $f(n, s)$ најмањи број новчића да се наплати износ s помоћу новчића који припадају префиксу дужине n полазног низа, тада важи

$$
\begin{aligned}
&f(n, 0) = 0, \\
&f(0, s) = +\infty, \quad \text{за } s > 0 \\
&f(n, s) = f(n-1, s), \quad \text{за } s < v_{n-1} \\
&f(n, s) = \min\left(f(n-1, s), 1 + f(n, s - v_{n-1})\right), \quad \text{за } s \geq v_{n-1}
\end{aligned}
$$

На основу овога је веома једноставно дефинисати рекурзивну функцију која израчунава тражени најмањи број новчића.

```
const int MAX_S = 2000;
const int MAX_V = 100;
const int INF = MAX_S + 1;

// najmanji broj novčića potreban da se naplati iznos S
// kada su nam na raspolaganju n vrednosti novčića datih u nizu v
int minBrojNovcica(int v[], int n, int s) {
  // iznos 0 se plaća sa 0 novčića
  if (s == 0)
    return 0;
  // ako nema novčića pozitivan iznos nije moguće platiti
  if (n == 0)
    return INF;
  // broj novčića ako se ne uzme ni jedan novčić poslednje vrste
  int br = minBrojNovcica(v, n-1, s);
  // ako je iznos veći od novčića poslednje vrste
  if (s >= v[n-1])
    // gledamo da li je bolje ako se uzme jedan novčić poslednje vrste
    br = min(br, minBrojNovcica(v, n, s - v[n-1]) + 1);
  // vraćamo rezultat
  return br;
}
```

Јасно је да у претходној функцији може доћи до понављања истих рекурзивних позива, што се може решити техником динамичког програмирања. Пошто функција има два променљива параметра, могуће је употребити матрицу за мемоизацију.

```
const int MAX_V = 100;
const int MAX_S = 2000;
const int INF = MAX_S + 1;

// matrica koju koristimo za memoizaciju
int memo[MAX_V][MAX_S + 1];

int minBrojNovcica(int v[], int n, int s){
  // ako smo već računali vrednost za (n, s), vraćamo upamćeni rezultat
  if (memo[n][s] != 0)
    return memo[n][s];
  // iznos 0 se naplaćuje sa 0 novčića
  if (s == 0)
    return memo[n][s] = 0;
  // ako nema novčića pozitivan iznos nije moguće platiti
  if (n == 0)
    return memo[n][s] = INF;
```

```
  // broj novčića ako se ne uzme ni jedan novčić poslednje vrste
  int br = minBrojNovcica(v, n-1, s);
  // ako je iznos veći od novčića poslednje vrste
  if (s >= v[n-1])
    // gledamo da li je bolje ako se uzme jedan novčić poslednje vrste
    br = min(br, minBrojNovcica(v, n, s - v[n-1]) + 1);
  // vraćamo rezultat
  return memo[n][s] = br;
}
```

Приликом динамичког програмирања навише, матрицу можемо попуњавати врсту по врсту и тако смањити меморијску сложеност.

Анализа сложености. Временска сложеност овог решења је $O(S \cdot N)$, где је S износ, а N број врста новчића, док је меморијска сложеност $O(S)$.

```
const int MAX_V = 100;
const int MAX_S = 2000;
const int INF = MAX_S + 1;


// najmanji broj novčića potreban da se naplati iznos S
// kada su nam na raspolaganju n vrednosti novčića datih u nizu v
int minBrojNovcica(int v[], int N, int S) {
  // najmanji broj novčića potrebnih da se plati svaki iznos od 0 do S
  int dp[MAX_S + 1];

  // analizu pokrećemo sa 0 vrsta novčića

  // iznos 0 se plaća sa 0 novčića
  dp[0] = 0;
  // ostale iznose nije moguće platiti
  for (int s = 1; s <= S; s++)
    dp[s] = INF;

  // uključujemo jednu po jednu vrstu novčića
  for (int n = 1; n <= N; n++) {
    // ažuriramo vrednosti svih iznosa
    for (int s = 0; s <= S; s++) {
      // ako je moguće uključiti novčić tekuće vrste
      if (s >= v[n-1])
        // ažuriramo minimum ako je to potrebno
        dp[s] = min(dp[s], dp[s - v[n-1]] + 1);
    }
  }

  // vraćamo najmanji broj novčića za iznos S
  return dp[S];
}
```

Анализа свих могућности за последњи новчић

Друго могуће решење разматра све могућности за последњи употребљени новчић. То може бити било који новчић који је мањи од текућег износа који треба наплатити, након чега се преостали износ и даље наплаћује помоћу свих могућих новчића. Ако са $f(s)$ обележимо најмањи број новчића потребних да се наплати износ s, при чему се могу користити сви расположиви новчићи, добијамо следећу рекурентну везу.

$$f(0) = 0,$$
$$f(s) = \min_{v_i \leq s} (1 + f(s - v_i))$$

Ако је износ позитиван, али мањи од свих новчића које имамо на располагању тада је минимум једнак $+\infty$. И у овом случају веома једноставно можемо дефинисати рекурзивну функцију.

```
const int MAX_S = 2000;
const int MAX_V = 100;
const int INF = MAX_S + 1;

// najmanji broj novčića potreban da se naplati iznos s
// kada su nam na raspolaganju n vrednosti novčića datih u nizu v
int minBrojNovcica(int v[], int n, int s) {
  // iznos 0 se naplaćuje sa 0 novčića
  if (s == 0)
    return 0;
  // minimalni broj novčića da se naplati iznos s (pretpostavljamo
  // da iznos nije moguće naplatiti)
  int br = INF;
  // razmatramo sve mogućnosti za poslednji novčić
  for (int i = 0; i < n; i++)
    // proveravamo da li je iznos s moguće naplatiti novčićem i
    if (v[i] <= s)
      // određujemo rekurzivno broj novčića za preostali iznos i
      // ažuriramo minimum ako je to potrebno
      br = min(br, minBrojNovcica(v, n, s-v[i]) + 1);
  // vraćamo rezultat
  return br;
}
```

Пошто долази до понављања истих рекурзивних позива потребно је да употребимо динамичко програмирање. Пошто је само један параметар променљив, за мемоизацију је довољно само да користимо низ.

```
const int MAX_S = 2000;
const int MAX_V = 100;
const int INF = MAX_S + 1;

// niz koji koristimo za memoizaciju - inicijalizovan podrazumevano na 0
int memo[MAX_S + 1];

// najmanji broj novčića potreban da se naplati iznos S
// kada su nam na raspolaganju n vrednosti novčića datih u nizu v
int minBrojNovcica(int v[], int n, int S) {
  // ako smo već izračunali broj novčića za iznos S, vraćamo upamćen rezultat
  if (memo[S] != 0)
    return memo[S];
  // iznos 0 se naplaćuje sa 0 novčića
  if (S == 0)
    return memo[S] = 0;
  // minimalni broj novčića da se naplati iznos s (pretpostavljamo
  // da iznos nije moguće naplatiti)
  int br = INF;
  // razmatramo sve mogućnosti za poslednji novčić
  for (int i = 0; i < n; i++)
    // proveravamo da li je iznos s moguće naplatiti novčićem i
    if (v[i] <= S)
      // određujemo rekurzivno broj novčića za preostali iznos i
```

```
      // ažuriramo minimum ako je to potrebno
      br = min(br, minBrojNovcica(v, n, S-v[i]) + 1);
  // vraćamo rezultat, pamteći ga pritom u nizu memo
  return memo[S] = br;
}
```

Приликом динамичког програмирања навише низ попуњавамо слева надесно.

```
const int MAX_S = 2000;
const int MAX_V = 100;
const int INF = MAX_S + 1;

// najmanji broj novčića potreban da se naplati iznos S
// kada su nam na raspolaganju n vrednosti novčića datih u nizu v
int minBrojNovcica(int v[], int n, int S) {
  int dp[MAX_S + 1];
  // iznos 0 se naplaćuje sa 0 novčića
  dp[0] = 0;
  // računamo minimalni broj novčića za sve ostale iznose
  for (int s = 1; s <= S; s++) {
    // minimalni broj novčića da se naplati iznos s (pretpostavljamo
    // da iznos nije moguće naplatiti)
    dp[s] = INF;
    // razmatramo sve mogućnosti za poslednji novčić
    for (int i = 0; i < n; i++)
      // proveravamo da li je iznos s moguće naplatiti novčićem i
      if (v[i] <= s)
        // ažuriramo minimum ako je to potrebno
        dp[s] = min(dp[s], dp[s-v[i]] + 1);
  }
  // vraćamo rezultat za iznos S
  return dp[S];
}
```

Задатак: Ранац 0-1

Програмер се сели из једне компаније у другу и жели да понесе предмете из своје старе канцеларије, међутим, на располагању само има један велики ранац у који можда не могу да стану сви предмети. Ако је позната маса и вредност сваког од предмета и ако је позната носивост ранца, напиши програм који одређује максималну вредност скупа предмета које програмер може да пренесе у ранцу.

Улаз: Са стандардног улаза се уноси носивост ранца (цео број између 1 и 150), затим број предмета n (цео број између 1 и 30), затим у наредних n редова масе и цене предмета (масе су цели бројеви између 1 и 10, а цене реални бројеви између $1,0$ и $100,0$, заокружени на две децимале).

Излаз: На стандардни излаз исписати највећу вредност предмета који се могу пренети у ранцу, заокружену на две децимале.

Пример 1

Улаз	*Излаз*
5 3 1 6.00 2 10.00 3 12.00	22.00

Објашњење

Највећа вредност се добије ако се пренесу предмети масе 2 и масе 3 (вредност је тада $10,0 + 12,0 = 22,0$).

Пример 2

Улаз

```
15
5
12 4.00
2 2.00
1 2.00
1 1.00
4 10.00
```

Излаз

```
15.00
```

Објашњење

Највећа вредност се добије ако се пренесу предмети маса 2, 1, 1 и 4 килограма (вредност је тада $2,0+2,0+1,0+10,0=15,0$).

Решење

Груба сила

Решење грубом силом подразумева да се испитају сви могући подскупови предмета и да се пронађе онај који може да стане у ранац а има највећу вредност. Сличан облик претраге подскупова ограниченог збира вршили смо у задатку Број поднизова датог збира.

Набрајање подскупова може да се изврши рекурзивном функцијом, чији је параметар дужина низа предмета n.

- Ако је низ предмета празан (ако је $n = 0$), тада је максимална вредност која се може спаковати једнака нули.
- Ако низ предмета није празан (ако је $n > 0$), тада анализирамо могућност да његов последњи елемент изоставимо или да га спакујемо у ранац. У првом случају рекурзивну функцију позивамо за исту вредност капацитета ранца и дужину низа $n-1$. Други случај је могућ само ако је маса последњег предмета мања од капацитета ранца. У том случају на резултат рекурзивног позива где је носивост ранца умањена за масу тог предмета и где је прослеђена дужина низа $n-1$ додајемо цену последњег предмета. Највећу вредност добијамо тако што израчунамо максимум два анализирана случаја (када последњи предмет није и када јесте стављен у ранац).

Ако са $f(n, W)$ обележимо максималну вредност која се може добити тако што се неки од првих n предмета спакују у ранац носивости W, важе следеће везе:

$$\begin{aligned} f(0,W) &= 0 \\ f(n,W) &= \max\left(f(n-1,W), f(n-1,W-w_{n-1})\right) + v_{n-1}, \quad \text{за } w_{n-1} \leq W \\ f(n,W) &= f(n-1,W), \quad \text{за } w_{n-1} > W \end{aligned}$$

Приметимо да рекурзивним позивима се тражи оптимум за скуп без последњег предмета, што је у реду, јер је за глобални оптимум неопходно и да су предмети из тог подскупа одабрани оптимално (задовољен је услов оптималне подструктуре).

Анализа сложености. У наивној рекурзивној имплементацији долази до преклапања идентичних рекурзивних позива, па је та имплементација веома неефикасна (сложеност најгорег случаја јој је експоненцијална).

```
double maxCena(const vector<int>& mase, const vector<double>& cene,
               double nosivost, int n) {
  if (n == 0)
     return 0.0;
  double cenaBez = maxCena(mase, cene, nosivost, n-1);
  if (mase[n-1] > nosivost)
     return cenaBez;
  double cenaSa = maxCena(mase, cene, nosivost - mase[n-1], n-1) + cene[n-1];
  return max(cenaBez, cenaSa);
}
```

Решење проблема поновљених рекурзивних позива, наравно, долази у облику динамичког програмирања (било мемоизације, било динамичког програмирања навише). Пошто имамо два променљива параметра (број предмета n и носивост ранца W), алоцирамо такву матрицу да свакој врсти одговара један префикс низа предмета, а свакој колони једна носивост. Матрицу можемо попуњавати врсту по врсту.

Пример. Прикажимо рад алгоритма на примеру када је носивост ранца 15 килограма и када имамо 5 предмета чије су масе и цене редом $(12, 4.00)$, $(2, 2.00)$, $(1, 2.00)$, $(1, 1.00)$ и $(4, 10.00)$.

```
    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
    ------------------------------------------------
0 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
1 | 0  0  0  0  0  0  0  0  0  0  0  0  4  4  4  4
2 | 0  0  2  2  2  2  2  2  2  2  2  2  4  4  6  6
3 | 0  2  2  4  4  4  4  4  4  4  4  4  4  6  6  8
4 | 0  2  3  4  5  5  5  5  5  5  5  5  5  6  7  8
5 | 0  2  3  2 10 12 13 14 15 15 15 15 15 15 15 15
```

Анализа сложености. Овим смо добили алгоритам чија је и временска и меморијска сложеност $O(n \cdot W)$ где је n број предмета, а W носивост ранца.

Приметимо да је веома важно било да је носивост ранца и да су масе предмета целобројне. Такође, обратимо пажњу на то да иако делује да смо овај проблем решили у полиномијалној сложености, то заправо није случај. Наиме, сложеност није изражена само у терминима величине улаза, већ у термину вредности на улазу (вредности носивости ранца). За овакве алгоритме се каже да су псеудо-полиномијални. Величина улаза везаног за број W одговара броју цифара броја W (нпр. броју бинарних цифара употребљених у запису), а време извршавања алгоритма експоненцијално расте у односу на тај број. За веће вредности W добили бисмо веома неефикасан алгоритам (и што се тиче утрошене меморије и што се тиче времена извршавања).

```
double maxCena(const vector<int>& mase, const vector<double>& cene,
               double nosivost, int n) {
  vector<vector<double>> dp(nosivost + 1);
  for (int M = 0; M <= nosivost; M++) {
    dp[M].resize(n+1);
    dp[M][0] = 0.0;
  }

  for (int N = 1; N <= n; N++) {
    for (int M = 0; M <= nosivost; M++) {
      dp[M][N] = dp[M][N-1];
      if (mase[N-1] <= M)
        dp[M][N] = max(dp[M][N-1], dp[M - mase[N-1]][N-1] + cene[N-1]);
    }
  }
  return dp[nosivost][n];
}
```

Можемо приметити да елементи сваке врсте зависе само од претходне, тако да не морамо чувати целу матрицу, већ само текућу врсту. Ажурирање врсте тада вршимо с њеног десног краја.

Анализа сложености. Овом оптимизацијом се меморијска сложеност смањује на $O(W)$, а временска сложеност остаје $O(n \cdot W)$. Приметимо да временска, али и меморијска сложеност остаје експоненцијална у односу на величину улаза (број битова потребних за запис вредности W).

```
double maxCena(const vector<int>& mase, const vector<double>& cene,
               double nosivost, int n) {
  vector<double> dp(nosivost + 1);
  dp[0] = 0.0;
  for (int N = 1; N <= n; N++) {
    for (int M = nosivost; M >= 0; M--)
      if (mase[N-1] <= M)
        dp[M] = max(dp[M], dp[M - mase[N-1]] + cene[N-1]);
  }
```

```
  return dp[nosivost];
}
```

Задатак: Едит растојање

Едит-растојање између две ниске се дефинише у терминима операција уметања, брисања и измена слова прве речи којима се може добити друга реч. Свака од ове три операције има своју цену. Дефинисати програм који израчунава најмању цену операција којима се од прве ниске може добити друга. На пример, ако је цена сваке операције јединична, тада се ниска `zdravo` може претворити у `bravo!` најефикасније операцијом измене слова `z` у `b`, брисања слова `d` и уметања карактера `!`.

Улаз: Са стандардног улаза се учитавају две ниске дужине највише 100 карактера, а затим цене операције уметања, брисања и измене (природни бројеви од 1 до 10, сваки у посебном реду).

Излаз: На стандардни излаз исписати тражену вредност едит-растојања.

Пример 1		**Пример 2**	
Улаз	*Излаз*	*Улаз*	*Излаз*
`zdravo`	`3`	`kitten`	`7`
`bravo!`		`sitting`	
`1`		`1`	
`1`		`2`	
`1`		`3`	

Решење

Рекурзивно решење

Изведимо прво индуктивно-рекурзивну конструкцију.

- Ако је прва ниска празна, најефикаснији начин да се од ње добије друга ниска је да се уметне један по један карактер друге ниске, тако да је минимална цена једнака производу цене операције уметања и броја карактера друге ниске.
- Ако је друга ниска празна, најефикаснији начин да се од прве ниске добије празна је да се један по један њен карактер избрише, тако да је минимална цена једнака производу цене операције брисања и броја карактера прве ниске.
- Индуктивна хипотеза ће бити да умемо да решимо проблем за било која два префикса прве и друге ниске. Ако су последња слова прве и друге ниске једнака, онда је потребно претворити префикс без последњег слова прве ниске у префикс без последњег слова друге ниске. Ако нису, онда имамо три могућности. Једна је да изменимо један од та два карактера у онај други и онда да, као у претходном случају, преведемо префиксе без последњих карактера један у други. Друга могућност је да обришемо последњи карактер прве ниске и пробамо да претворимо тако њен добијени префикс у другу ниску. Трећа могућност је да прву ниску трансформишемо у префикс друге ниске без последњег карактера и да затим додамо последњи карактер друге ниске.

На основу овога лако можемо дефинисати рекурзивну функцију која израчунава едит-растојање. Да нам се ниске не би мењале током рекурзије (што може бити споро), ефикасније је да ниске прослеђујемо у неизмењеном облику и да само прослеђујемо бројеве карактера њихових префикса који се тренутно разматрају.

Анализа сложености. У директној рекурзивној имплементацији долази до великог броја поновљених рекурзивних позива, што доводи до веома лоше (експоненцијалне) сложености.

```
int editRastojanje(const string& s1, const string& s2, int n1, int n2) {
  if (n1 == 0)
    return n2 * cenaUmetanja;
  if (n2 == 0)
    return n1 * cenaBrisanja;
  if (s1[n1-1] == s2[n2-1])
    return editRastojanje(s1, s2, n1-1, n2-1);
  int r1 = editRastojanje(s1, s2, n1-1, n2) + cenaUmetanja;
  int r2 = editRastojanje(s1, s2, n1, n2-1) + cenaBrisanja;
  int r3 = editRastojanje(s1, s2, n1-1, n2-1) + cenaIzmene;
  return min({r1, r2, r3});
```

```
}

int editRastojanje(const string& s1, const string& s2) {
  int n1 = s1.size(), n2 = s2.size();
  return editRastojanje(s1, s2, n1, n2);
}
```

Динамичко програмирање навише

Решење директном рекурзијом је, наравно, изразито неефикасно због преклапајућих рекурзивних позива. Алгоритам динамичког програмирања навише за овај проблем познат је под именом Вагнер-Фишеров алгоритам. Резултате за префиксе дужине i и j памтићемо у матрици на пољу (i, j). Дакле, ако су дужине ниски n_1 и n_2, потребна нам је матрица димензије $(n_1 + 1) \times (n_2 + 1)$, а коначан резултат ће се налазити на месту (n_1, n_2). Ако матрицу попуњавамо врсту по врсту, слева надесно, приликом израчунавања елемента на позицији (i, j), биће израчунати сви елементи матрице од којег он зависи (а то су $(i - 1, j - 1)$, $(i - 1, j)$ и $(i, j - 1)$).

Пример. Под претпоставком да су цене јединичне, за ниске `zdravo` и `bravo!` добија се следећа матрица.

```
    b r a v o !
   0 1 2 3 4 5 6
   -------------
 0|0 1 2 3 4 5 6
z1|1 1 2 3 4 5 6
d2|2 2 2 3 4 5 6
r3|3 3 2 3 4 5 6
a4|4 4 3 2 3 4 5
v5|5 5 4 3 2 3 4
o6|6 6 5 4 3 2 3
```

Анализа сложености. Пошто се током рада алгоритма попуњава матрица димензије $(n_1 + 1) \times (n_2 + 1)$, а свако поље матрице се попуњава у времену $O(1)$, укупна временска и меморијска сложеност алгоритма је $O(n_1 \cdot n_2)$.

```
int editRastojanje(const string& s1, const string& s2,
                   int cenaUmetanja, int cenaBrisanja, int cenaIzmene) {
  int n1 = s1.size(), n2 = s2.size();
  vector<vector<int>> dp(n1+1);
  for (int i = 0; i <= n1; i++)
    dp[i].resize(n2+1);

  dp[0][0] = 0;
  for (int i = 0; i <= n1; i++)
    dp[i][0] = i * cenaBrisanja;
  for (int j = 0; j <= n2; j++)
    dp[0][j] = j * cenaUmetanja;
  for (int i = 1; i <= n1; i++)
    for (int j = 1; j <= n2; j++) {
      if (s1[i-1] == s2[j-1])
        dp[i][j] = dp[i-1][j-1];
      else {
        int r1 = dp[i-1][j] + cenaUmetanja;
        int r2 = dp[i][j-1] + cenaBrisanja;
        int r3 = dp[i-1][j-1] + cenaIzmene;
        dp[i][j] = min({r1, r2, r3});
      }
    }

  return dp[n1][n2];
}
```

Меморијска оптимизација

На основу поставке задатка, није потребно одредити саме измене, већ само растојање (на пример, ако се врши провера да ли су две ниске блиске приликом претраге у којој се допушта да је корисник направио и неколико словних грешака). Пошто елементи текућег реда зависе само од претходног, можемо извршити меморијску оптимизацију и истовремено чувати само један ред. Током ажурирања елемента на позицији j његов део на позицијама строго мањим од j ће чувати елементе текућег реда i, део од позиције j надаље ће чувати елементе претходног реда $i-1$. Променљива `prethodni` ће чувати вредност са поља $(i-1, j-1)$, а променљива `tekuci` ће чувати вредност са поља $(i-1, j)$.

Анализа сложености. Временска сложеност након ове оптимизације је $O(n_1 \cdot n_2)$, док се меморијска сложеност може спустити на $O(\min(n_1, n_2))$ (тако што се одабере да ли ће се попуњавати врста по врста или колона по колона).

```
int editRastojanje(const string& s1, const string& s2,
                   int cenaUmetanja, int cenaBrisanja, int cenaIzmene) {
  int n1 = s1.size(), n2 = s2.size();
  vector<int> dp(n2 + 1);
  dp[0] = 0;
  for (int j = 0; j <= n2; j++)
    dp[j] = j * cenaUmetanja;
  for (int i = 1; i <= n1; i++) {
    int prethodni = dp[0];
    dp[0] = i * cenaBrisanja;
    for (int j = 1; j <= n2; j++) {
      int tekuci = dp[j];
      if (s1[i-1] == s2[j-1])
        dp[j] = prethodni;
      else {
        int r1 = tekuci + cenaUmetanja;
        int r2 = dp[j-1] + cenaBrisanja;
        int r3 = prethodni + cenaIzmene;
        dp[j] = min({r1, r2, r3});
      }
      prethodni = tekuci;
    }
  }
  return dp[n2];
}
```

Реконструкција оптималног пута

На основу попуњене матрице лако је реконструисати и сам низ корака који прву ниску трансформише у другу. Крећемо од доњег десног угла матрице и крећемо се уназад. У сваком кораку проверавамо како смо дошли на текућу позицију и у складу са тим корак убацујемо у низ (вектор). На крају, када стигнемо до горњег левог угла, низ корака исписујемо уназад.

Пример. У текућем примеру на поље (6, 6) смо стигли са поља (6, 5) што значи да је последњи корак уметање карактера `!`. На поље (6, 5) смо стигли са поља (5, 4) при чему није вршен никаква измена. Слично, на поље (5, 4) смо стигли са (4, 3), на поље (4, 3) смо стигли са (3, 2), а на поље (3, 2) смо стигли са (2, 1). На поље (2, 1) смо могли стићи са (1, 1) при чему је обрисан каракер `d`, а на поље (1, 1) смо стигли са (0, 0) тако што је карактер `z` промењен у `b`. Дакле, један низ могућих корака је

```
zdravo
          измена z у b
bdravo
          брисање d
bravo
          уметање !
bravo!
```

Приметимо да смо на поље (2, 1) могли стићи и са поља (1, 0) операцијом измене `d` у `b`. На поље (1, 0) смо стигли са (0, 0) операцијом брисања слова `z`. На тај начин добијамо следећи низ корака.

```
zdravo
         брисање z
dravo
         измена d у b
bravo
         уметање !
bravo!
```

Имплементација функције којом се врши реконструкција (једног) решења може бити следећа.

```
void ispisiIzmene(const vector<vector<int>>& dp,
                  const string& s1, const string& s2,
                  int cenaUmetanja, int cenaBrisanja, int cenaIzmene) {
  vector<string> izmene;
  int n1 = s1.size(), n2 = s2.size();
  while (n1 > 0 || n2 > 0) {
    cout << n1 << " " << n2 << endl;
    if (n1 > 0 && n2 > 0 && dp[n1][n2] == dp[n1-1][n2-1] && s1[n1-1] == s2[n2-1]) {
      n1--; n2--;
    } else if (n1 > 0 && n2 > 0 && dp[n1][n2] == dp[n1-1][n2-1] + cenaIzmene) {
      izmene.push_back(string("Izmena: ") + s1[n1-1] + " -> " + s2[n2-1]);
      n1--; n2--;
    } else if (n2 > 0 && dp[n1][n2] == dp[n1][n2-1] + cenaUmetanja) {
      izmene.push_back(string("Umetanje: ") + s2[n2-1]);
      n2--;
    } else if (n1 > 0 && dp[n1][n2] == dp[n1-1][n2] + cenaBrisanja) {
      izmene.push_back(string("Brisanje: ") + s1[n1-1]);
      n1--;
    }
  }
  for (auto it = izmene.rbegin(); it != izmene.rend(); it++)
    cout << *it << endl;
}
```

Задатак: Најдужи заједнички подниз две ниске

Напиши програм који израчунава дужину највећег заједничког подниза две ниске. Подниз чине карактери ниске који не морају бити узастопни, али се јављају у истом редоследу као у оригиналној ниски. На пример за ниске `abacbc` и `babbca` најдужа заједничка подниска је `babc`.

Улаз: Две линије стандардног улаза садрже две ниске које се састоје од малих слова енглеске азбуке и дугачке су највише 1000 карактера.

Излаз: На стандардни излаз исписати само тражену дужину.

Пример

Улаз	*Излаз*
`xmjyauz` `mzjawxu`	`4`

Решење

Рекурзивно решење

Ако је било која од две ниске празна, тада је једини њен подниз празан, па је дужина најдужег заједничког подниза једнака нули.

Ако су обе ниске непразне, тада можемо упоредити њихова последња слова.

- Ако су она једнака, она могу бити укључена у најдужи заједнички подниз две ниске и проблем се рекурзивно своди на проналажење најдужег заједничког подниза префикса тих ниски добијених искључива-

њем последњих слова. Нагласимо да није грешка експлицитно анализирати и случајеве када неко од та два последња слова није укључено у најдужи заједнички подниз (тиме смо сигурнији да ћемо добити коректан алгоритам), али се може доказати да за тим нема потребе.

- У супротном, није могуће да оба последња слова буду укључена у заједнички подниз. Зато разматрамо најдужи заједнички подниз прве ниске и префикса друге ниске без њеног последњег слова и заједнички подниз друге ниске и префикса прве ниске без њеног последњег слова. Дужи од два подниза биће најдужи заједнички подниз те две ниске. Нагласимо и да у овом случају није неопходно експлицитно анализирати најдужи заједнички подниз та два префикса (јер он не може бити дужи од најдужих заједничких поднизова добијених када се неки од тих префикса прошири додатним словом).

Пошто рекурзија тече по префиксима ниски, једини променљиви параметри током рекурзије могу бити дужине тих префикса. Ако са $f(n_1, n_2)$ означимо дужину најдужег заједничког подниза префикса ниске s дужине n_1 и префикса ниске t дужине n_2, тада важи следећа рекурентна веза:

$$
\begin{aligned}
f(0, n_2) &= 0 \\
f(n_1, 0) &= 0 \\
f(n_1, n_2) &= f(n_1 - 1, n_2 - 1) + 1, \quad \text{за } s_{n_1-1} = t_{n_2-1} \\
f(n_1, n_2) &= \max\left(f(n_1, n_2 - 1), f(n_1 - 1, n_2)\right), \quad \text{за } s_{n_1-1} \neq t_{n_2-1}
\end{aligned}
$$

```
int najduziZajednickiPodniz(const string& s1, int n1,
                            const string& s2, int n2) {
    if (n1 == 0 || n2 == 0)
       return 0;
    int rez = max(najduziZajednickiPodniz(s1, n1, s2, n2-1),
                  najduziZajednickiPodniz(s1, n1-1, s2, n2));
    if (s1[n1-1] == s2[n2-1])
       rez = max(rez, najduziZajednickiPodniz(s1, n1-1, s2, n2-1) + 1);
    return rez;
}

int najduziZajednickiPodniz(const string& s1, const string& s2) {
  return najduziZajednickiPodniz(s1, s1.size(), s2, s2.size());
}
```

Мемоизација

У директном рекурзивном решењу има много преклапајућих рекурзивних позива. Стога је ефикасност могуће поправити техником динамичког програмирања. Један могући приступ је да употребимо мемоизацију. Вредност дужине најдужег подниза за сваки пар дужина префикса можемо памтити у матрици.

```
int najduziZajednickiPodniz(const string& s1, int n1,
                            const string& s2, int n2,
                            vector<vector<int>>& memo) {
  if (memo[n1][n2] != -1)
    return memo[n1][n2];

  if (n1 == 0 || n2 == 0)
    return memo[n1][n2] = 0;
  int rez = max(najduziZajednickiPodniz(s1, n1, s2, n2-1, memo),
                najduziZajednickiPodniz(s1, n1-1, s2, n2, memo));
  if (s1[n1-1] == s2[n2-1])
    rez = max(rez, najduziZajednickiPodniz(s1, n1-1, s2, n2-1, memo) + 1);
  return memo[n1][n2] = rez;
}

int najduziZajednickiPodniz(const string& s1, const string& s2) {
  int n1 = s1.size(), n2 = s2.size();
  vector<vector<int>> memo(n1+1);
```

```
  for (int i = 0; i <= n1; i++)
    memo[i].resize(n2 + 1, -1);

  return najduziZajednickiPodniz(s1, n1, s2, n2, memo);
}
```

Динамичко програмирање навише

Проблем прекалпајућих рекурзивних позива се може решити ако се употреби динамичко програмирање навише. Дужине најдужих поднизова префикса можемо чувати у матрици. Елемент матрице на позицији (n_1, n_2) зависи само од елемената на позицијама $(n_1 - 1, n_2)$, $(n_1, n_2 - 1)$ и $(n_1 - 1, n_2 - 1)$, тако да матрицу пожемо да попуњавамо било врсту по врсту, било колону по колону.

Пример. Прикажимо матрицу за пример две ниске `xmjyauz` и `mzjawxu`.

```
    mzjawxu
   01234567
  ---------
 0|00000000
x 1|00000011
m 2|01111111
j 3|01122222
y 4|01122222
a 5|01123333
u 6|01123334
z 7|01223334
```

```
int najduziZajednickiPodniz(const string& s1, const string& s2) {
  int n1 = s1.size(), n2 = s2.size();
  vector<vector<int>> dp(n1+1);
  for (int i = 0; i <= n1; i++)
    dp[i].resize(n2 + 1);

  for (int i = 0; i <= n1; i++)
    dp[i][0] = 0;
  for (int j = 0; j <= n2; j++)
    dp[0][j] = 0;

  for (int i = 1; i <= n1; i++)
    for (int j = 1; j <= n2; j++) {
      dp[i][j] = max(dp[i][j-1], dp[i-1][j]);
      if (s1[i-1] == s2[j-1])
        dp[i][j] = max(dp[i][j], dp[i-1][j-1] + 1);
    }

  return dp[n1][n2];
}
```

Меморијска оптимизација

Можемо приметити да се приликом попуњавања матрице врсту по врсту садржај сваке наредне врсте попуњава само на основу претходне врсте. Стога није потребно истовремено памтити целу матрицу, већ је довољно памтити само једну, текућу врсту. Ажурирање врсте морамо вршити с лева надесно, јер сваки елемент у текућој врсти зависи од елемента који му претходи у тој врсти. Приметимо да нам је у неком тренутку потребно да знамо претходни елемент текуће врсте, а понекад претходни елемент претходне врсте, тако да приликом ажурирања врсте морамо да у помоћној променљивој памтимо стару вредност претходног елемента врста (јер се ажурирањем претходног елемента његова стара вредност губи, а она нам може затребати у случају да су одговарајући карактери у нискама једнаки).

```
int najduziZajednickiPodniz(const string& s1, const string& s2) {
  int n1 = s1.size(), n2 = s2.size();
```

```
  vector<int> dp(n2 + 1, 0);
  for (int i = 1; i <= n1; i++) {
    int prethodni = dp[0];
    for (int j = 1; j <= n2; j++) {
      int tekuci = dp[j];
      if (s1[i-1] == s2[j-1])
        dp[j] = prethodni + 1;
      else
        dp[j] = max(dp[j-1], dp[j]);
      prethodni = tekuci;
    }
  }
  return dp[n2];
}
```

Задатак: Најдужи растући подниз

Напиши програм који одређује најдужи строго растући подниз (не обавезно узастопних елемената) унутар датог низа.

Улаз: Са стандардног улаза се учитава број елемената низа n ($1 \leq n \leq 50000$), а затим елементи низа (цели бројеви, сваки у посебном реду).

Излаз: На стандардни излаз исписати дужину најдужег растућег подниза.

Пример

Улаз	*Излаз*
10 3 2 6 9 5 4 3 7 2 8	4

Објашњење

Један растући подниз дужине 4 је `2 6 7 8`.

Решење

Приказаћемо два решења заснована на динамичком програмирању, која су различите ефикасности.

Најдужи растући подниз који се завршава на датој позицији у низу

У првој групи решења разматраћемо позицију по позицију у низу и одредићемо најдужи растући подниз чији је последњи елемент на свакој од њих. Најдужи растући подниз се онда добити као најдужи од свих тих поднизова (јер се он сигурно завршава на некој позицији у низу). Приликом одређивања дужине најдужег растућег подниза који се завршава на позицији $i \geq 0$, претпоставићемо да за сваку претходну позицију (ако их има) умемо да одредимо дужину најдужег растућег подниза који се на њој завршава. Низ који се завршава на позицији i сигурно садржи елемент a_i, а може продужити све оне низове који се завршавају на некој позицији $0 \leq j < i$ ако је $a_j < a_i$. Да би низ који се завршава на позицији i био што дужи, његов префикс који се завршава на позицији j мора бити што дужи (а дужину најдужег растућег низа за сваку позицију j можемо одредити рекурзивно). Зато од свих низова који се завршавају на позицијама j, таквим да је $a_j < a_i$ одређујемо најдужи и продужавамо га елементом a_i (ако таквих елемената нема, тада је најдужи низ који се завршава на позицији a_i једночлан).

Анализа сложености. У директној рекурзивној имплементацији долази од преклапања рекурзивних позива, што доводи до веома неефикасног решења, експоненцијалне сложености.

```
// pronalazi duzinu najduzeg strogo rastuceg podniza niza a koji se
// zavrsava elementom na poziciji i
int najduziRastuciPodniz(const vector<int>& a, int i) {
  int maksI = 1;
  for (int j = 0; j < i; j++)
    if (a[j] < a[i]) {
      int maksJ = najduziRastuciPodniz(a, j);
      if (maksJ + 1 > maksI)
        maksI = maksJ + 1;
    }
  return maksI;
}

// pronalazi duzinu najduzeg strogo rastuceg podniza niza a
int najduziRastuciPodniz(const vector<int>& a) {
  int maks = 0;
  for (int i = 0; i < a.size(); i++) {
    int maksI = najduziRastuciPodniz(a, i);
    if (maksI > maks)
      maks = maksI;
  }
  return maks;
}
```

Мемоизација

Неефикасност која настаје услед понављања идентичних рекурзивних позива решавамо динамичким програмирањем. За мемоизацију довољно је да памтимо низ дужина најдужих растућих низова који се завршавају на свакој позицији у низу. Пошто су све те дужине веће или једнаке од 1 (сваки елемент сам за себе чини растући низ), низ у који меморишемо решења можемо иницијализовати нулама (што значи да је тражена дужина још непозната).

```
// pronalazi duzinu najduzeg strogo rastuceg podniza niza a koji se
// zavrsava elementom na poziciji i
int najduziRastuciPodniz(const vector<int>& a, int i,
                         vector<int>& memo) {
  if (memo[i] != 0)
    return memo[i];
  int maksI = 1;
  for (int j = 0; j < i; j++) {
    if (a[j] < a[i]) {
      int maksJ = najduziRastuciPodniz(a, j, memo);
      if (maksJ + 1 > maksI)
        maksI = maksJ + 1;
    }
  }
  return memo[i] = maksI;
}

// pronalazi duzinu najduzeg strogo rastuceg podniza niza a
int najduziRastuciPodniz(const vector<int>& a) {
  vector<int> memo(a.size(), 0);
  int maks = 0;
  for (int i = 0; i < a.size(); i++) {
    int maksI = najduziRastuciPodniz(a, i, memo);
    if (maksI > maks)
      maks = maksI;
  }
  return maks;
}
```

Динамичко програмирање навише

Једноставно можемо формулисати и динамичко програмирање навише, тако што низ попуњавамо слева надесно. Након попуњавања низа одређујемо његов максимум. Нагласимо да сваки наредни елемент низа потенцијално зависи од великог броја претходних тако да није могуће редуковати меморијску сложеност тиме што би се памтили само неки елементи низа (морамо увек знати дужине свих претходних елемената).

Пример. Прикажимо на примеру како ће се тај низ попуњавати.

```
i  0 1 2 3 4 5 6 7 8 9
ai 3 2 6 9 5 4 3 7 8 2
dp 1 1 2 3 2 2 2 3 4 1
```

На пример, када израчунавамо елемент на позицији 5 анализирамо низове који се завршавају елементима мањим од вредности 4 која се налази на позицији 5. То су вредности 3 на позицији 0 и 2 на позицији 1. У оба случаја максимална дужина подниза који се завршава на тој позицији је 1, па се било који од тих низова продужава елементом 4 и добија се растући низ дужине 2.

Анализа сложености. Решење које се добија на овај начин је меморијске сложености $O(n)$ и временске сложености $O(n^2)$.

```
// pronalazi duzinu najduzeg strogo rastuceg podniza niza a
int najduzi_rastuci_podniz(const vector<int>& a) {
  int n = a.size();
  vector<int> dp(n);
  for (int i = 0; i < n; i++) {
    dp[i] = 1;
    for (int j = 0; j < i; j++)
      if (a[i] > a[j] && dp[j] + 1 > dp[i])
        dp[i] = dp[j] + 1;
  }

  int max = dp[0];
  for (int i = 0; i < n; i++)
    if (dp[i] > max)
      max = dp[i];
  return max;
}
```

Најмањи елемент којим се завршава растући подниз дате дужине

Изменом индуктивно-рекурзивне конструкције можемо добити и много ефикасније решење. Кључна идеја је да претпоставимо да уз дужину d_{max} најдужег растућег подниза до сада обрађеног дела низа можемо за сваку дужину подниза $1 \leq d \leq d_{max}$ да одредимо најмањи елемент којим се завршава неки растући подниз дужине d. Приметимо да низ тих вредности увек строго растући (ако постоји строго растући низ дужине d који се завршава неким елементом a_i, тада се његов префикс дужине $d-1$ мора завршавати неким елементом који је строго мањи од елемента a_i).

Пример. Низ обрађујемо елемент по елемент. Размотримо један пример (колоне табеле одговарају дужинама низа), а елементи низа који се обрађују су написани десно.

```
1 2 3 4 5 6 7 8 9 10
- - - - - - - - - -
3 - - - - - - - - -         3
2 - - - - - - - - -         2
2 6 - - - - - - - -         6
2 6 9 - - - - - - -         9
2 5 9 - - - - - - -         5
2 4 9 - - - - - - -         4
2 3 9 - - - - - - -         3
2 3 7 - - - - - - -         7
2 3 7 - - - - - - -         2
2 3 7 8 - - - - - -         8
```

Ако се досадашњи најдужи подниз завршавао елементом који је мањи од текућег, онда смо нашли подниз који је дужи за један и најмањи елемент на крају тог подниза је текући. То се у примеру дешава приликом обраде елемента 3, елемента 6, елемента 9 и елемента 8.

Размотримо ситуацију у којој обрађујемо елемент 5. До тада смо видели елементе 3, 2, 6 и 9. Елемент 2 на првој позицији у табели означава да је најмањи елемент којим се може завршити једночлани растући низ једнак 2. Елемент 6 на другој позицији у табели означава да је најмањи елемент којим се може завршити двочлани растући низ једнак 6 (у питању је низ 2 6 или низ 3 6). Елемент 9 на трећој позицији у табели означава да је најмањи елемент којим се може завршити трочлани растући низ једнак 9 (у питању је низ 2 6 9 или 3 6 9). Пошто је 5 мањи од 9 ниједан од ових трочланих низова није могуће проширити елементом 5, па је четворочланих растућих низова нема. Поставља се питање да ли се можда трочлани низови могу завршити елементом 5, но ни то није могуће. Наиме, пошто је у табели двочланим низовима придружена вредност 6, то значи да се сви двочлани растући низови завршавају бар са 6, па није могуће 9 заменити са 5. Са друге стране, пошто је 5 веће од 2, завршни елемент двочланих низова 6 је могуће заменити са 5 и тиме добити мању завршну вредност двочланих низова (то су у овом случају низови 3 5 и 2 5). Дакле, у табели вредност 6 треба заменити вредношћу 5. Вредност 2 лево од 6 нема смисла заменити са 5, јер би се тиме завршна вредност једночланих низова увећала, а ми у табели памтимо најмање завршне вредности.

На основу анализе овог примера можемо да закључимо да је приликом анализе сваког текућег елемента потребно пронаћи прву позицију d у табели на којој се налази елемент који је већи или једнак од текућег и позицију d уместо тога уписати текући елемент. Ако су сви елементи мањи од текућег (ако је $d = d_{max}$), онда се текући елемент додаје на крај низа (и у том случају заправо радимо исто - уписујемо елемент на позицију d). Остали елементи у табели остају непромењени. Заиста на свим позицијама у табели лево од позиције d уписани су елементи строго мањи од текућег и њиховом заменом са текућим се не би смањила вредност завршног елемената тих низова. За елементе десно од позиције d, иако су већи од текућег, ажурирање није могуће. У свим низовима дужине $d' > d$ неки префикс се морао завршавати елементом на позицији d или елементом већим од њега, а пошто је он био већи или једнак од текућег, заменог последњег елемента текућим не бисмо добили више растући низ.

Кључни добитак настаје када се примети да, пошто су елементи у табели сортирани, позицију првог елемента који је већи или једнак од текућег можемо остварити бинарном претрагом. Отуда следи ефикасна имплементација (у низу dp вредност најмањег завршног елемента за низове дужине d памтимо на позицији $d - 1$).

Анализа сложености. Временска сложеност такве имплементације је $O(n \log(n))$, док је меморијска сложеност $O(n)$.

Бинарна претрага може бити извршена библиотечком функцијом.

```
// pronalazi duzinu najduzeg strogo rastuceg podniza niza a
int najduziRastuciPodniz(const vector<int>& a) {
  int n = a.size();
  vector<int> dp(n);
  int max = 0;
  for (int i = 0; i < n; i++) {
    auto it = lower_bound(dp.begin(), next(dp.begin(), max), a[i]);
    *it = a[i];
    int d = distance(dp.begin(), it);
    if (d + 1 > max)
      max = d + 1;
  }
  return max;
}
```

Још једна могућност је да се проналазак првог елемента већег или једанког датом коришћењем бинарне претраге имплементира ручно.

```
int prviVeciIliJednak(const vector<int>& a, int l, int d, int x) {
  while (l < d) {
    int s = l + (d - l) / 2;
    if (a[s] < x)
      l = s + 1;
    else
```

```
      d = s;
  }
  return l;
}

int najduziRastuciPodniz(const vector<int>& a) {
  int n = a.size();
  vector<int> dp(n+1);
  int max = 0;
  for (int i = 0; i < n; i++) {
    int k = prviVeciIliJednak(dp, 0, max, a[i]);
    dp[k] = a[i];
    if (k + 1 > max)
      max = k + 1;
  }
  return max;
}
```

Свођење на проблем најдужег заједничког подниза

Постоји веома једноставно свођење овог проблема на проблем проналажења најдужег заједничког подниза два низа. Решење тог проблема већ приказано је у задатку Најдужи заједнички подниз две ниске. Наиме, дужина најдужег растућег подниза датог низа једнака је дужини најдужег заједничког подниза тог низа и низа који се добија неопадајућим сортирањем и уклањањем дупликата тог низа.

```
int najduziZajednickiPodniz(const vector<int>& s1, const vector<int>& s2) {
  int n1 = s1.size(), n2 = s2.size();
  vector<int> dp(n2 + 1, 0);
  for (int i = 1; i <= n1; i++) {
    int prethodni = dp[0];
    for (int j = 1; j <= n2; j++) {
      int tekuci = dp[j];
      if (s1[i-1] == s2[j-1]) {
        dp[j] = prethodni + 1;
      } else {
        dp[j] = max(dp[j-1], dp[j]);
      }
      prethodni = tekuci;
    }
  }
  return dp[n2];
}

int najduziRastuciPodniz(const vector<int>& a) {
  // sortirana kopija niza a
  vector<int> b = a;
  sort(begin(b), end(b));
  // uklanjamo duplikate iz vektora b
  b.erase(unique(begin(b), end(b)), end(b));
  return najduziZajednickiPodniz(a, b);
}
```

Задатак: Оптимално множење матрица

Множење матрица димензије $D_1 \times D_2$ и димензије $D_2 \times D_3$ даје матрицу димензије $D_1 \times D_3$ и да би се оно спровело потребно је $D_1 \cdot D_2 \cdot D_3$ множења бројева. Када је потребно измножити дужи низ матрица, онда ефикасност зависи од начина како се те матрице групишу (множење је асоцијативна операција и допуштено је било које груписање множења). Напиши програм који одређује најмањи број множења бројева потребан да би се измножио низ матрица датих димензија.

Улаз: Са стандардног улаза се прво учитава број $3 \leq n \leq 100$, а затим и низ димензија $D_0, D_1, D_2, \ldots D_{n-1}$. Матрице које се множе су димензија $D_0 \times D_1$, $D_1 \times D_2$, ..., $D_{n-2} \times D_{n-1}$.

Излаз: На стандардни излаз исписати тражени најмањи број множења.

Пример

Улаз	*Излаз*
4 5 4 6 2	88

Објашњење: Множимо матрицу A димензије 5×4, матрицу B димензије 4×6 и матрицу C димензије 6×2. За рачунање производа $(AB)C$ потребно је $5 \cdot 4 \cdot 6 + 5 \cdot 6 \cdot 2 = 180$ операција, а за рачунање производа $A(BC)$ потребно је $4 \cdot 6 \cdot 2 + 5 \cdot 4 \cdot 2 = 88$ операција множења бројева.

Решење

Рекурзивно решење

Кренимо од рекурзивног решења. Како год да групишемо матрице неко множење је то које се последње извршава. То може бити множење прве матрице и производа свих осталих, множење производа прве две матрице и производа осталих матрица, множење производа прве три матрице и производа осталих матрица итд. све до множења производа свих матрица пре последње последњом матрицом. Основна идеја је да експлицитно изанализирамо све те могућности и да одаберемо најбољу од њих. За сваки фиксирани избор позиције последњег множења потребно је одредити како множити све матрице лево и све матрице десно од те позиције. Кључни увид је да је да би глобални избор био оптималан и та два потпроблема потребно решити на оптималан начин (јер у случају да не одаберемо оптимални распоред заграда у неком потпроблему, боље глобално решење можемо добити заменом тог распореда оптималним). Дакле, потпроблеме можемо решавати рекурзивним позивима.

Нека $f(l, d)$ означава минималан број множења потребан да се измноже матрице димензија $D_l, \ldots, D_d$. Потребно је одредити $f(0, n-1)$. На основу претходне дискусије, важе следеће рекурентне везе.

$$f(l,d) = 0, \quad \text{за } d - l + 1 \leq 2$$
$$f(l,d) = \min_{l<i<d} (f(l,i) + f(i,d) + D_l \cdot D_i \cdot D_d), \quad \text{за } d - l + 1 \geq 3$$

Заиста, ако је $d - l + 1 \leq 2$, то значи да се ради само о једној матрици и није потребно вршити било каква множења.

У супротном постоји више од једне матрице. Позиција последњег множења може бити свака позиција строго већа од l и строго мања од d. Када је последње множење на позицији i значи да се на крају множи производ матрица димензија $D_l, \ldots, D_i$ и производ матрица димензија $D_i, \ldots, D_d$. Рекурзивно одређујемо минималан број множења за сваки од тих производа. Први производ даје матрицу димензије $D_l \times D_i$, други даје матрицу димензије $D_i \times D_d$, и на крају се врши множење те две матрице, за шта је потребно додатних $D_l \times D_i \times D_d$ операција.

```
long long minBrojMnozenja(const vector<int>& dimenzije, int l, int d) {
  int n = d - l + 1;
  if (n <= 2)
    return 0;
  long long min = numeric_limits<long long>::max();
  for (int i = l+1; i <= d-1; i++) {
    long long broj = minBrojMnozenja(dimenzije, l, i) +
                     minBrojMnozenja(dimenzije, i, d) +
                     dimenzije[l] * dimenzije[i] * dimenzije[d];
    if (broj < min)
      min = broj;
  }
```

```
  return min;
}

long long minBrojMnozenja(const vector<int>& dimenzije) {
  return minBrojMnozenja(dimenzije, 0, dimenzije.size() - 1);
}
```

Мемоизација

Директно рекурзивно решење доводи до великог броја идентичних рекурзивних позива и неупоредиво боље решење се добија динамичким програмирањем. Најједноставније решење је додати мемоизацију рекурзивној функцији. Пошто функција има два променљива целобројна параметра (то су l и d), чије се вредности крећу у интервалу $[0, n-1]$, мемоизацију можемо извршити помоћу матрице димензије $n \times n$. Коначно решење се налази на позицији $(0, n-1)$.

```
long long minBrojMnozenja(const vector<int>& dimenzije, int l, int d,
                          vector<vector<long long>>& memo) {
  if (memo[l][d] != -1)
    return memo[l][d];
  int n = d - l + 1;
  if (n == 2)
    return 0;
  long long min = numeric_limits<long long>::max();
  for (int i = l+1; i <= d-1; i++) {
    long long broj = minBrojMnozenja(dimenzije, l, i, memo) +
                     minBrojMnozenja(dimenzije, i, d, memo) +
                     dimenzije[l] * dimenzije[i] * dimenzije[d];
    if (broj < min)
      min = broj;
  }
  return memo[l][d] = min;
}

long long minBrojMnozenja(const vector<int>& dimenzije) {
  int n = dimenzije.size();
  vector<vector<long long>> memo(n);
  for (int i = 0; i < n; i++)
    memo[i].resize(n, -1);
  return minBrojMnozenja(dimenzije, 0, dimenzije.size() - 1, memo);
}
```

Динамичко програмирање навише

Динамичко програмирање навише је донекле компликованије, због компликованијих зависности између елемената матрице. Прво, пошто важи да је $l \leq d$, релевантан нам је само део матрице изнад њене главне дијагонале. Попуњавање није могуће ни по врстама, ни по колонама. Може се уочити да сваки елемент зависи само од елемената који се налазе испод дијагонале на којој се тај елемент налази. Зато је елементе могуће израчунавати редом, по дијагоналама. За $d - l + 1 \leq 2$, важи да је $f(l, d) = 0$, па елементе на главној дијагонали и дијагонали инзад ње постављамо на нулу, а затим рачунамо елементе на дијагоналама изнад њих. Сваку дијагоналу карактерише константни размак између индекса l и d тј. исти број елемената у интервалу $[l, d]$ тј. исти број матрица које се множе.

Пример. Прикажимо на једном примеру како се гради и попуњава матрица. Нека су матрице димензија 4, 3, 5, 1, 2.

```
     0  1  2  3  4
------------------
0 |  0  0 60 27 35
1 |  0  0  0 15 21
2 |  0  0  0  0 10
```

```
3 |  0  0  0  0  0
4 |  0  0  0  0  0
```

- Вредност на пољу $(0, 2)$ подразумева да се множе матрице димензија 4×3 и 3×5, за шта је потребно $4 \cdot 3 \cdot 5 = 60$ операција множења.
- Вредност на пољу $(1, 3)$ подразумева да се множе матрице димензија 3×5 и 5×1, за шта је потребно $3 \cdot 5 \cdot 1 = 15$ операција множења.
- Вредност на пољу $(2, 4)$ подразумева да се множе матрице димензија 5×1 и 1×2, за шта је потребно $5 \cdot 1 \cdot 2 = 10$ операција множења.
- Вредност на пољу $(0, 3)$, подразумева да се множе матрице димензије 4×3, 3×5 и 5×1.

  Једна, боља, могућност је да се прво помноже матрице димензије 3×5 и 5×1 за шта је потребно 15 операција множења, а да се онда прва матрица димензије 4×3 помножи добијеном матрицом димензије 3×1, за шта је потребно додатних $4 \cdot 3 \cdot 1 = 12$ операција множења.

  Друга могућност је да се прво помноже прве две матрице, за шта је потребно 60 операција множења, а да се затим добијена матрица димензије 4×5 помножи матрицом димензије 5×1, за шта је потребно додатних $4 \cdot 5 \cdot 1 = 20$ операција множења.
- Вредност на пољу $(1, 4)$, подразумева да се множе матрице димензије 3×5, 5×1 и 1×2.

  Једна могућност је да се прво помноже матрице димензије 5×1 и 1×2 за шта је потребно 10 операција множења, а да се онда прва матрица димензије 3×5 помножи добијеном матрицом димензије 5×2, за шта је потребно додатних $3 \cdot 5 \cdot 2 = 30$ операција множења.

  Друга, боља, могућност је да се прво помноже прве две матрице, за шта је потребно 15 операција множења, а да се затим добијена матрица димензије 3×1 помножи матрицом димензије 1×2, за шта је потребно додатних $3 \cdot 1 \cdot 2 = 6$ операција множења.
- Вредност на пољу $(0, 4)$, подразумева да се множе матрице димензије 4×3, 3×5, 5×1 и 1×2.

  Једна могућност је да се прва матрица димензије 4×3 помножи производом осталих матрица, за који знамо да се може израчунати помоћу 21 операције множења. Након тога је потребно помножити матрицу димензије 4×3 и добијену матрицу димензије 3×2, за шта је потребно додатних $4 \cdot 3 \cdot 2 = 24$ операције множења.

  Друга могућност је да се производ матрица димензија 4×3 и 3×5, који се израчунава помоћу 60 операција множења помножи производом матрица димензија 5×1 и 1×2, који се израчунава помоћу 10 операција множења. Производ добијених матрица димензија 4×5 и 5×2 захтева још $4 \cdot 5 \cdot 2 = 40$ операција множења.

  Трећа, најбоља, могућност је да се производ прве три матрице, који се израчунава помоћу 27 операција множења помножи последњом матрицом димензије 1×2. Пошто је производ прве три матрице димензије 4×1, ово захтева додатних $4 \cdot 1 \cdot 2 = 8$ операција множења.

Анализом зависности између елемената матрице може се установити да није могуће једноставно смањивање меморијске сложености, као што је то био случај у неким раније приказаним проблемима.

```
long long minBrojMnozenja(const vector<int>& dimenzije) {
  int n = dimenzije.size();
  vector<vector<long long>> dp(n);
  for (int i = 0; i < n; i++)
    dp[i].resize(n, 0);

  for (int k = 3; k <= n; k++)
    for (int l = 0, d = l + k - 1; d < n; l++, d++) {
      dp[l][d] = numeric_limits<long long>::max();
      for (int i = l+1; i <= d-1; i++) {
        long broj = dp[l][i] + dp[i][d] +
          dimenzije[l] * dimenzije[i] * dimenzije[d];
        if (broj < dp[l][d])
          dp[l][d] = broj;
      }
```

```
    }

  return dp[0][n-1];
}
```

Реконструкција оптималног редоследа множења

Да би се реконструисало решење, можемо у посебној матрици памтити позиције минималних елемената.

```
void odstampaj(const vector<vector<int>>& poz, int l, int d) {
  int n = d - l + 1;
  if (n <= 2)
    cout << "A" << l;
  else {
    cout << "(";
    odstampaj(poz, l, poz[l][d]);
    cout << "*";
    odstampaj(poz, poz[l][d], d);
    cout << ")";
  }
}

long long minBrojMnozenja(const vector<int>& dimenzije) {
  int n = dimenzije.size();
  vector<vector<long long>> dp(n);
  for (int i = 0; i < n; i++)
    dp[i].resize(n, 0);

  vector<vector<int>> poz(n);
  for (int i = 0; i < n; i++)
    poz[i].resize(n);


  for (int k = 3; k <= n; k++)
    for (int l = 0, d = l + k - 1; d < n; l++, d++) {
      dp[l][d] = numeric_limits<long long>::max();
      for (int i = l+1; i <= d-1; i++) {
        int broj = dp[l][i] + dp[i][d] +
          dimenzije[l] * dimenzije[i] * dimenzije[d];
        if (broj < dp[l][d]) {
          dp[l][d] = broj;
          poz[l][d] = i;
        }
      }
    }
  odstampaj(poz, 0, n-1);
  cout << endl;
  return dp[0][n-1];
}
```

Задатак: Најдужи подниз палиндром

Написати програм којим се за дати стрингу s одређује дужину најдужег подниза ниске s који је палиндром. Подниз не мора да садржи узастопне карактере ниске, али они морају да се јављају у истом редоследу (подниз се добија брисањем произвољног броја карактера).

Улаз: Са стандардног улаза се учитава ниска s састављена само од малих слова енглеске абецеде, чија је дужина највише 5000 карактера.

Излаз: На стандардни излаз исписати само тражену дужину најдужег палиндромског подниза.

Пример

Улаз	*Излаз*
`najduzipalindrom`	`5`

Решење

Рекурзивно решење

Кренимо од рекурзивног решења.

- Празна ниска има само празан подниз, па је дужина најдужег палиндромског подниза једнака нули. Ниска дужине 1 је сама свој палиндромски подниз, па је дужина њеног најдужег палиндромског подниза једнака 1.
- Ако ниска има бар два карактера, онда рамзатрамо да ли су њен први и последњи карактер једнаки. Ако јесу, онда они могу бити део најдужег палиндромског подниза и проблем се своди на проналажење најдужег палиндромског подниза дела ниске без првог и последњег карактера. У супротном они не могу истовремено бити део најдужег палиндромског подниза и потребно је елиминисати бар један од њих. Проблем, дакле, сводимо на проналажење најдужег палиндромског подниза суфикса ниске без првог карактера и на проналажење најдужег палиндромског подниза префикса ниске без последњег карактера. Дужи од та два палиндромска подниза је тражени палиндромски подниз целе ниске.

Овим је практично дефинисана рекурзивна процедура којом се решава проблем. У сваком рекурзивном позиву врши се анализа неког сегмента (низа узастопних карактера полазне ниске), па је сваки рекурзивни позив одређен са два броја који представљају границе тог сегмента. Ако са $f(l, d)$ означимо дужину најдужег палиндромског подниза дела ниске $s[l, d]$, тада важе следеће рекурентне везе.

$$
\begin{aligned}
f(l,d) &= 0, \quad \text{за } l > d \\
f(l,d) &= 1, \quad \text{за } l = d \\
f(l,d) &= 2 + f(l+1, d-1), \quad \text{за } l < d \text{ и } s_l = s_d \\
f(l,d) &= \max\left(f(l+1,d), f(l,d-1)\right), \quad \text{за } l < d \text{ и } s_l \neq s_d
\end{aligned}
$$

На основу овога, функцију је веома једноставно имплементирати.

```
int najduziPalindrom(const string& s, int l, int d) {
  if (l > d)
    return 0;
  if (l == d)
    return 1;
  if (s[l] == s[d])
    return 2 + najduziPalindrom(s, l+1, d-1);
  return max(najduziPalindrom(s, l, d-1),
             najduziPalindrom(s, l+1, d));
}

int najduziPalindrom(const string& s) {
  return najduziPalindrom(s, 0, s.length() - 1);
}
```

Анализа сложености. У овој имплементацији долази до преклапања рекурзивних позива, па је она веома неефикасна (сложеност најгорег случаја је експоненцијална).

Мемоизација

У претходној функцији долази до преклапања рекурзивних позива, па је пожељно употребити мемоизацију. За мемоизацију користимо матрицу (практично, њен горњи троугао у којем је $l < d$).

```
int najduziPalindrom(const string& s, int l, int d,
                     vector<vector<int>>& memo) {
  if (memo[l][d] != -1)
```

```
    return memo[l][d];
  if (l > d)
    return memo[l][d] = 0;
  if (l == d)
    return memo[l][d] = 1;
  if (s[l] == s[d])
    return memo[l][d] = 2 + najduziPalindrom(s, l+1, d-1, memo);
  return memo[l][d] = max(najduziPalindrom(s, l, d-1, memo),
                          najduziPalindrom(s, l+1, d, memo));
}

int najduziPalindrom(const string& s) {
  vector<vector<int>> memo(s.length(), vector<int>(s.length(), -1));
  return najduziPalindrom(s, 0, s.length() - 1, memo);
}
```

Динамичко програмирање навише

До ефикасног решења можемо доћи и динамичким програмирањем одоздо навише. Елемент на позицији (l, d) матрице зависи од елемената на позицијама $(l+1, d)$, $(l, d-1)$ и $(l+1, d-1)$, док се коначно решење налази у горњем левом углу матрице, тј. на пољу $(0, n-1)$. Због оваквих зависности матрицу не можемо попуњавати ни врсту по врсту, ни колону по колону, већ дијагоналу по дијагоналу. На дијагоналу испод главне уписујемо све нуле, на главну дијагоналу све јединице, а затим попуњавамо једну по једну дијагоналу изнад главне, све док не дођемо до елемента у горњем левом углу.

Пример. Прикажимо како изгледа попуњена матрица на примеру ниске `abaccba`.

```
    abaccba
    0123456
    -------
a 0|1133346
b 1|0111244
a 2| 011224
c 3|  01222
c 4|   0111
b 5|    011
a 6|     01
```

Коначно решење 6 одговара поднизу `abccba`.

```
int najduziPalindrom(const string& s) {
  int n = s.length();
  vector<vector<int>> dp(n);
  for (int i = 0; i < n; i++) {
    dp[i].resize(n, 0);
    dp[i][i] = 1;
  }
  for (int k = 1; k < n; k++)
    for (int l = 0; l + k < n; l++) {
      int d = l + k;
      if (s[l] == s[d])
        dp[l][d] = dp[l+1][d-1] + 2;
      else
        dp[l][d] = max(dp[l+1][d], dp[l][d-1]);
    }

  return dp[0][n - 1];
}
```

Анализа сложености. Ово решење има и меморијску и временску сложеност $O(n^2)$.

Меморијска оптимизација

Примећујемо да елементи сваке дијагонале зависе само од елемената претходне две дијагонале. Могуће је да чувамо само две дијагонале - текућу и претходну. Током ажурирања текуће дијагонале њене постојеће елементе истовремено преписујемо у претходну. Када су карактери једнаки, тада у привремену променљиву бележимо одговарајући елемент претходне дијагонале, на његово место уписујемо одговарајући елемент текуће дијагонале, а онда на место тог елемента уписујемо вредност привремене променљиве увећану за два. Када су карактери различити одговарајући елемент текуће дијагонале уписујемо на одговарајуће место у претходној дијагонали, а на његово место уписујемо максимум те и наредне вредности текуће дијагонале.

```
int najduziPalindrom(const string& s) {
  int n = s.length();
  // elementi dve prethodne dijagonale
  vector<int> dpp(n, 0);
  vector<int> dp(n, 1);
  for (int k = 1; k < n; k++) {
    for (int l = 0; l + k < n; l++) {
      int d = l + k;
      if (s[l] == s[d]) {
        int tmp = dp[l];
        dp[l] = dpp[l+1] + 2;
        dpp[l] = tmp;
      }
      else {
        dpp[l] = dp[l];
        dp[l] = max(dp[l], dp[l+1]);
      }
    }
    dpp[n-k] = dp[n-k];
  }

  return dp[0];
}
```

Анализа сложености. Меморијска сложеност овог решења је $O(n)$, док временска сложеност остаје $O(n^2)$.

Свођење на проблем најдужег заједничког подниза две ниске

Рецимо још и да је решење овог задатка могуће добити и свођењем на проблем одређивања најдужег заједничког подниза две ниске. Тај је проблем описан у задатку Најдужи заједнички подниз две ниске. Наиме, најдужи палиндромски подниз једнак је најдужем заједничком поднизу оригиналне ниске и ниске која се добија њеним обртањем.

```
int najduziZajednickiPodniz(const string& s1, const string& s2) {
  int n1 = s1.size(), n2 = s2.size();
  vector<int> dp(n2 + 1, 0);
  for (int i = 1; i <= n1; i++) {
    int prethodni = dp[0];
    for (int j = 1; j <= n2; j++) {
      int tekuci = dp[j];
      if (s1[i-1] == s2[j-1]) {
        dp[j] = prethodni + 1;
      } else {
        dp[j] = max(dp[j-1], dp[j]);
      }
      prethodni = tekuci;
    }
  }
  return dp[n2];
}
```

```
int najduziPalindrom(const string& s) {
  string sObratno = s;
  reverse(begin(sObratno), end(sObratno));
  return najduziZajednickiPodniz(s, sObratno);
}
```

Анализа сложености. Сложеност ове редукције је иста као и сложеност директног решења (временски $O(n^2)$, а просторно $O(n)$).