Глава 5

Подели па владај

Основни механизам конструкције алгоритама је тзв. индуктивно-рекурзивна конструкција где се решење проблема своди на решавање потпроблема истог облика, али мање димензије. Веома често је димензија потпроблема за један мања од димензије оригиналног проблема. На пример, низ се разлаже на свој први елемент и низ елемената иза њега (суфикс) или се разлаже на елементе који претходе последњем елементу (префикс) и тај последњи елемент. На пример, у алгоритму сортирања селекцијом (selection sort) на прво место се поставља најмањи елемент низа, а затим се на исти начин обрађују елементи иза њега, док се у алгоритму сортирања уметањем (insertion sort) сортира префикс низа у који се на крају умеће последњи елемент. Ови алгоритми су описани у задатку Сортирање бројева. Осим решавања потпроблема, алгоритам обично садржи кораке потребне да се потпроблем припреми и кораке да се од резултата потпроблема добију резултати полазног проблема. На пример, у алгоритму сортирања селекцијом припремни корак је одређивање позиције минимума и његово довођење на почетак низа, док је у алгоритму сортирања уметањем након обраде попт-проблема (сортирања префикса) додатни корак уметање последњег елемента. Ако се, као што је то случај у ова два примера, обрада потпроблема врши било као први или као последњи корак алгоритма (ако не постоји фаза припреме или обраде резултата потпроблема), тада имплементација може бити и итеративна.

- Ако се решава један потпроблем и ако се припрема потпроблема и обрада резултата потпроблема врше у времену $O(1)$, уз уобичајену претпоставку да се излаз из рекурзије такође врши у времену $O(1)$, време решавања задовољава једначину $T(n) = T(n-1) + O(1), T(0) = O(1)$, чије је решење $T(n) = O(n)$.
- Ако се решава један потпроблем и ако се припрема потпроблема и обрада резултата потпроблема врше у времену $O(n)$, уз уобичајену претпоставку да се излаз из рекурзије такође врши у времену $O(1)$, време решавања задовољава једначину $T(n) = T(n-1) + O(n), T(0) = O(1)$, чије је решење $T(n) = O(n^2)$.

Ефикаснија решења се често добијају техником која се назива техника *разлагања*, техника *декомпозиције* или техника *подели-па-владај* (енгл. divide-and-conquer). Њена основна идеја је то да је често ефикасније решавати неколико потпроблема чија је димензија два (или више) пута мања од димензије полазног потпроблема. Таквих проблема може бити два или више. Код изразито једноставних проблема решење је могуће добити и решавањем само једног таквог потпроблема, чиме се добијају изразито ефикасна решења.

Основни примере технике разлагања су сортирање обједињавањем (енгл. merge sort) и брзо сортирање (енгл. quick sort). И ови су алгоритми описани у задатку Сортирање бројева. Такође, алгоритми обраде бинарног дрвета спадају у ову групу (јер су поддрвета потпроблеми који су у случају балансираних дрвета два пута мањи од димензије полазног проблема). Бинарна претрага се такође може убројати у алгоритам овог типа, али, пошто се код ње један потпроблем потпуно занемарује (примењује се одсецање), та техника се понекад назива *смањи па владај* (енгл. decrease and conquer).

Ако су потпроблеми који се решавају једнаке димензије (два пута мање од димензије полазног проблема), добија се нека од следећих једначина (у зависности од тога колико времена је потребно за припрему потпроблема и обједињавање резултата добијених потпроблема). Решења једначина овог типа се могу често добити применом *мастер теореме*, а ми ћемо у наставку резимирати само неколико најкарактеристичнијих.

- Једначина $T(n) = 2T(n/2) + O(n), T(0) = O(1)$ има решење $O(n \log n)$.
- Једначина $T(n) = 2T(n/2) + O(1), T(0) = O(1)$ има решење $O(n)$.
- Једначина $T(n) = 2T(n/2) + O(\log n), T(0) = O(1)$ има решење $O(n)$.

- Једначина $T(n) = 2T(n/2) + O(n \log n), T(0) = O(1)$ има решење $O(n \log^2(n))$.

Треба обратити пажњу на то да ако су потпроблеми стално неравномерне димензије, могуће је да се добије процес који се дегенерише у процес који се описује једначином $T(n) = (n-1) + (n), T(0) = O(1)$ и чије је решење (n^2) (што је, на пример, сложеност најгорег случаја у алгоритму quick sort, ако пивот

У случају алгоритама типа *decrease and conquer*, решава се само један потпроблем и добијају се најчешће једначине следећег типа.

- Једначина $T(n) = T(n/2) + O(n), T(0) = O(1)$ има решење $O(n)$.
- Једначина $T(n) = T(n/2) + O(1), T(0) = O(1)$ има решење $O(\log n)$.

Задатак: Сортирање бројева

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Брзо сортирање (QuickSort)

У сваком кораку алгоритма сортирања један елемент (обично називан *пивот*) се доводи на своје место (пожељно близу средине низа). Да би након тога, проблем могао бити сведен на сортирање два мања подниза, потребно је приликом довођења пивота на своје место груписати све елементе мање или једнаке од њега лево од њега, а све елементе веће од њега десно од њега (ако се низ сортира неопадајуће). То прегруписавање елемената низа, *корак партиционисања* кључни је корак алгоритма брзог сотирања.

Брзо сортирање се може имплементирати на следећи начин. Позив `qsort_(a, l, d)` sortira deo niza $a[l, d]$. Партиционисање се врши техником два показивача. Слична техника је приказана у задацима Двобојка и Тробојка.

Након партиционисање рекурзивно се сортирају лева и десна половина низа. Излаз из рекурзије представља случај када је низ (тј. његов део $a[l, d]$) празан или једночлан (такав низ је већ сортиран).

Сложеност најгорег случаја овог алгоритма може бити квадратна тј. $O(n^2)$, ако се стално дешава да пивот дели низ на две неравномерне целине. Ипак, може се доказати да је просечна сложеност овог алгоритма $O(n \log n)$ и у пракси он показује веома добре резултате (за разлику од сортирања обједињавањем не троши се време на померање елемената између помоћног и главног низа).

```
// soritra segment pozicija [l, d] u nizu a
void quick_sort(vector<int>& a, int l, int d) {
  // ako segment [l, d] jedan ili nula elementa on je vec sortiran
  if (l < d) {
    // za pivot uzimamo proizvoljan element segmenta
    swap(a[l], a[l + rand() % (d - l + 1)]);
    // particionisemo niz tako da se u njemu prvo javljaju elementi
    // manji ili jednaki pivotu, a zatim veci od pivota
    // tokom rada vazi [l, k] su manji ili jednaki pivotu
    // (k, i) su veci od pivota, [i, d] su jos neispitani
    int k = l;
    for (int i = l+1; i <= d; i++)
      if (a[i] <= a[l])
        swap(a[i], a[++k]);
    // razmenjujemo pivot sa poslednjim manjim ili jednakim elementom
    swap(a[l], a[k]);
    // rekurzivno sortiramo deo niza levo i desno od pivota
    quick_sort(a, l, k - 1);
    quick_sort(a, k + 1, d);
  }
}
```

```
// sortira niz a
void quick_sort(vector<int>& a) {
  // poziv pomocne funkcije koja u nizu a sortira segment pozicija [0, n-1]
  quick_sort(a, 0, a.size() - 1);
}
```

Види другачија решења овог задатка.

Задатак: Збир k најбољих

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

QuickSelect

Алгоритам *брзе селекције* (QuickSelect) представља модификацију алгоритма брзог сортирања. Алгоритам брзог сортирања и његова имплементација су приказани у задатку Сортирање бројева. Брза селекција се користи да се низ подели тако да се на првих k места нађе k највећих (или најмањих) елемената низа, да се на местима иза њих налазе елементи мањи (или већи) од њих, при чему је редослед елемената у свакој од те две групе произвољан. Пошто је редослед елемената у групама произвољан, може се добити ефикаснији алгоритам од онога у ком би се сортирао цео низ (јер редослед елемената у свакој групи може бити произвољан).

Алгоритам брзе селекције се заснива на кораку партиционисања, идентичном као у алгоритму брзог сортирања, који у линеарној сложености елементе низа уређује тако да се прво у низу нађу елементи који су мањи од неког датог елемента (тзв. пивота), да се након њих налази тај елемент и да након тога следе елементи који су већи од пивота. Редослед елемената у свакој од ових група је потпуно произвољан. Ако се пивот јавља више пута, остала појављивања пивота могу бити било лево, било десно од пивота (често се узима да се лево од пивота налазе елементи мањи или једнаки од њега, а десно елементи строго већи од њега или обратно). Као код алгоритма брзог сортирања, за партиционисање се могу користити поступци засновани на техници два показивача. Детаљан опис поступака овог типа дат је у задацима Двобојка или Тробојка.

Пошто се у задатку тражи збир k највећих елемената у низу, једноставности ради, претпоставићемо да елементе низа сортирамо у обратном редоследу – желимо да се k највећих елемената низа нађе на његовом почетку.

Осовни алгоритам је рекурзиван и параметар рекурзије су границе l и d и број k, и задатак алгоритма је да део низа на позицијама $[l, d]$ реорганизује тако да на почетку буде k највећих елемената тог дела низа.

Излаз из рекурзије може бити када је k веће или једнако од дужине сегмента $[l, d]$, тј. када је $k \geq d - l + 1$ тада сви елементи низа спадају међу првих k. Ако је у старту $k \leq n$, где је n број елемената низа, тада k никада неће бити строго веће од дужине сегмента $[l, d]$ (али може бити једнако).

Након избора пивота и партиционисања, позната је позиција на којој се пивот налази. Нека је то нека позиција m (важи да је $l \leq m \leq d$).

- Ако је број елемената лево од пивота (вредност $m - l$) већа или једнака k онда је довољно наћи k највећих елемената левог дела низа. Наиме сви елементи у левом делу низа су већи или једанки од пивота и од елемената у десном делу, па се свих k највећих елемената дела низа са позиција $[l, d]$ налазе у левом делу низа, испред пивота. Дакле, потребно је извршити рекурзивни позив за интервал $[l, m-1]$ и број k тј. пронаћи k највећих елемената у делу низа на позицијама $[l, m-1]$.
- У супротном се закључно са пивотом налази $m - l + 1$ од k највећих елемената низа и зато је потребно у десном делу одредити још $k - (m - l + 1)$ највећих елемената из тог дела, тако да се рекурзивни позив врши за интервал $[m + 1, d]$ и број $k - m + l - 1$.

Приметимо да у функцији постоји само један рекурзивни позив и то репни, тако да се он може једноставно елиминисати.

Прикажимо рад овог алгоритма на примеру проналажења 5 највећих елемената низа $5, 9, 6, 3, 10, 13, 1, 7, 8, 14, 2$. Важи да је $k = 5$ и $n = 11$.

- На почетку је $[l, d] = [0, n-1] = [0, 10]$. После партиционисања добија се низ $9, 6, 10, 13, 7, 8, 14, \mathit{5}, 3, 1, 2$, где је пивот 5 завршио на позицији $m = 7$. Пошто је број елемената лево од пивота $m - l = 7$ већи од $k = 5$, рекурзивно проналазимо 5 највећих елемената у делу низа на позицијама у интервалу $[l, m-1] = [0, 6]$.
- Партиционишемо део низа одређен са $[l, d] = [0, 6]$. После партиционисања добија се низ $10, 13, 14, \mathit{9}, 6, 7, 8, \mathbf{5}, \mathbf{3}, \mathbf{1}, \mathbf{2}$, где је пивот 9 завршио на позицији $m = 3$. Овај пут је број елемената лево од пивота $m - l = 3$ строго мањи од $k = 5$. Зато је потребно рекурзивно понаћи $k - (m - l + 1) = 1$ највећи елемент у делу низа на позицијама $[m + 1, d] = [4, 6]$
- Партиционишемо део низа одређен са $[l, d] = [4, 6]$. После партиционисања добија се низ $\mathbf{10}, \mathbf{13}, \mathbf{14}, \mathbf{9}, 7, 8, \mathit{6}, \mathbf{5}, \mathbf{3}, \mathbf{1}, \mathbf{2}$, где је пивот 6 завршио на позицији $m = 6$. Број елемената лево од пивота $m - l = 2$ и он је већи од k, па рекурзивно тражимо $k = 1$ највећи елемент у делу низа на позицијама $[l, m-1] = [4, 5]$.
- Партиционишемо део низа одређен са $[l, d] = [4, 5]$. После партиционисања добија се низ $\mathbf{10}, \mathbf{13}, \mathbf{14}, \mathbf{9}, 8, 7, \mathbf{6}, \mathbf{5}, \mathbf{3}, \mathbf{1}, \mathbf{2}$, где је пивот 7 завршио на позицији $m = 5$. Број елемената лево од пивота $m - l = 1$ и он је једнак k, па рекурзивно тражимо $k = 1$ највећи елемент у делу низа на позицијама $[l, m-1] = [4, 4]$.
- Пошто је дужина $d - l + 1$ интервала $[l, d] = [4, 4]$ једнака $k = 1$, поступак је завршен. Коначно стање низа је $10, 13, 14, 9, 8, 7, 6, 5, 3, 1, 2$. Приметимо да низ није сортиран опадајуће, али смо сигурни да се 5 највећих елемената низа сада налази на његовом почетку.

Приметимо и да доказивање заустављања алгоритма није сасвим тривијално. Ако је k у старту веће од дужине низа, алгоритам ће се одмах зауставити. У супротном ће све време извршавања алгоритма важити инваријанта да је $0 \leq k \leq d - l + 1$ (у тренутку када се постигне горња једнакост, алгоритам ће се зауставити), док ће се вредност $d - l + 1$ смањивати кроз рекурзивне позиве.

- Ако је $k \leq m - l$, извршиће се рекурзивни позив за $l' = l$, $d' = m - 1$ и $k' = k$. Пошто је $m \leq d$, важи и да је $m < d + 1$. Зато је $d' - l' + 1 = m - 1 - l + 1 = m - l < d - l + 1$.

  Важи и да је $k' \leq d' - l' + 1$, јер је $k' = k \leq m - l = d' - l' + 1$. Важи и да је $0 \leq k' = k$.

- Ако је $k > m - l$, извршиће се рекурзивни позив за $l' = m + 1$, $d' = d$ и $k' = k - (m - l + 1)$. Тада је $d' - l' + 1 = d - (m+1) + 1 = d - m$. Пошто је $l \leq m$, важи да је $d' - l' + 1 = d - m \leq d - l < d - l + 1$.

  Важи и да је $k' \leq d' - l' + 1$. Заменом добијамо да је тај услов еквивалентан $k - (m - l + 1) \leq d - (m + 1) + 1$, тј. $k + l - 1 \leq d$, но то важи, јер је $k \leq d - l + 1$.

  Пошто је $k > m - l$, важи и да је $k \geq m - l + 1$, па је $k' = k - (m - l + 1) \geq 0$.

Из ове анализе јасно је и да ако је $k \leq n$, тада услов заустављања алгоритма (излаз из рекурзије) може бити и када низ постане празан тј. када је $l - d + 1$ постане 0. Такође, услов заустављања би могао бити и то да је $k = 0$.

Под претпоставком да ће пивот делити низ на делове који су отприлике једнаке величине, сложеност овог алгоритма се описује једначином $T(n) = T(n/2) + O(n), T(0) = O(1)$, чије је решење $T(n) = O(n)$. Дакле, сложеност алгоритма брзе селекције је линеарна $O(n)$. Пошто се цео низ учитава и чува у меморији и просторна сложеност је $O(n)$.

```
// QuickSelect - odredjujemo najvecih k elemenata niza a tj. niz permutujemo
// tako da se najvecih k elemenata nadju na prvih k pozicija (u proizvoljnom
// redosledu)
void qsortK(vector<int>& a, int l, int d, int k) {
  if (k >= d - l + 1)
    return;

  // niz particionisemo tako da se pivot (element a[l]) dovede na
  // svoje mesto, da ispred njega budu svi elementi koji su veci ili
  // jednaki od njega, a da iza njega budu svi elementi veci od njega
  int m = l;
  for (int t = l+1; t <= d; t++)
    if (a[t] >= a[l])
      swap(a[++m], a[t]);
```

```
  swap(a[m], a[l]);

  if (k <= m - l)
    // svih k elemenata su levo od pivota - obradjujemo deo ispred pivota
    qsortK(a, l, m - 1, k);
  else
    // mozda su neki kod k najvecih iza pivota - obradjujemo deo iza pivota
    qsortK(a, m+1, d, k - (m - l + 1));
}

// QuickSelect - pomocna funkcija zbog lepseg interfejsa
void qsortK(vector<int>& a, int k) {
  qsortK(a, 0, a.size() - 1, k);
}
```

Рачунање збира током партиционисања

Уместо да елементе прво распоредимо тако да k највећих елемената буде на почетку, па тек затим да их сабирамо, функција може бити дефинисана тако да истовремено распоређује елементе и рачуна њихов збир. У овом случају згодно нам је да за излаз из рекурзије прогласимо услов $k = 0$ или услов да је низ празан тј. да је $d - l + 1 = 0$, јер у оба случаја функција треба да врати збир нула, што се веома једноставно програмира (ако би излаз из рекурзије био да k постане једнако дужини низа, приликом излаза из рекурзије, требало би израчунати збир свих елемената низа, што је мало компликованије).

Пошто је рекурзија репна, она се лако елиминише.

```
// QuickSelect - zbir k najvecih elemenata niza
int zbirKNajvecih(vector<int>& a, int k) {
  int l = 0, d = a.size() - 1;
  int zbir = 0;
  while (k != 0) {
    // niz particionisemo tako da se pivot (element a[l]) dovede na
    // svoje mesto, da ispred njega budu svi elementi koji su veci ili
    // jednaki od njega, a da iza njega budu svi elementi veci od njega
    int m = l;
    for (int t = l+1; t <= d; t++)
      if (a[t] >= a[l])
        swap(a[++m], a[t]);
    swap(a[m], a[l]);

    if (k <= m - l)
      // svih k elemenata su levo od pivota - obradjujemo deo ispred pivota
      d = m - 1;
    else {
      // sabiramo sve elemente zakljucno sa pivotom
      for (int i = l; i <= m; i++)
        zbir += a[i];
      // mozda su neki kod k najvecih iza pivota - obradjujemo deo iza pivota
      k -= m - l + 1;
      l = m + 1;
    }
  }
  return zbir;
}
```

Библиотечка функција

У језику C++ библиотечка функција `nth_element` врши поделу низа тако да се на позицији n нађе елемент који ту и припада у сортираном редоследу, да се испред те позиције нађу елементи који су сви мањи или једнаки од њега, а да се иза те позиције нађу елементи који су сви већи или једнаки од њега. Функцији се

прослеђује итератор на почетак дела низа (вектора) који се обрађује (обично добијен помоћу `begin`), итератор на неку позицију на средини низа и итератор који указује непосредно иза краја низа (обично добијен помоћу `end`). Ако средишњи итератор указује на n-ту позицију у низу након примене функције на тој позицији ће се наћи n-ти по величини елемент, док ће сви елементи лево од њега бити мањи или једнаки од свих елемената десно од њега. Рецимо и да постоји функција `partial_sort` која је слична претходној али уједно елементе испред дате позиције уређује (соритра) по величини, међутим, у овом случају то нам није потребно и тиме би се само непотребно губило време.

```
// niz particionisemo tako da je  k-ti element na svom mestu i da su
// svi elementi ispred njega manji ili jednaki od svih elemenata iza
nth_element(a.begin(), next(a.begin(), k), a.end(), greater<int>());

// odredjujemo i ispisujemo zbir prvih k elemenata transformisanog niza
cout << accumulate(a.begin(), next(a.begin(), k), 0) << endl;
```

Задатак: Сортирање бројева

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Сортирање обједињавањем (MergeSort)

Алгоритам сортирања обједињавањем дели низ на два дела чије се дужине разликују највише за 1 (уколико је дужина низа паран број, онда су ова два дела једнаких дужина), рекурзивно сортира сваки од њих и затим обједињује сортиране половине. За обједињавање је неопходно користити додатни, помоћни низ, а на крају се обједињени низ копира у полазни низ. Излаз из рекурзије је случај једночланог низа (случај празног низа не може да наступи осим ако је полазни низ празан).

Кључна операција у овом алгоритму је операција обједињавања сортираних низова, техником два показивача. Њена имплементација описана је у задатку Обједињавање. На пример, обједињавањем сортираних низова `a` и `b` добија се сортирани низ `c`. Два већ сортирана низа могу се објединити у трећи сортирани низ само једним проласком кроз низове (тј. у линеарном времену $O(m+n)$ где су m и n димензије полазних низова).

```
a:  1 3 4 7 9 11                  b:  2 5 8 9 10 12 14
         c: 1 2 3 4 5 7 8 9 9 10 11 12 14
```

Функција сортирања обједињавањем сортира део низа $a[l, d]$, уз коришћење низа tmp као помоћног. Променљива n чува број елемената који се сортирају у оквиру овог рекурзивног позива, а променљива s чува средишњи индекс у низу између l и d. Рекурзивно се сортира $n_1 = \lfloor \frac{n}{2} \rfloor$ елемената између позиција l и $s-1$ и $n_2 = n - \lfloor \frac{n}{2} \rfloor$ елемената између позиција s і d. Након тога, сортирани поднизови обједињују се у помоћни низ. Пошто се више не обједињују цели низови, већ делови једног низа, функцију обједињавања морамо мало прилагодити.

Помоћни низ може се пре почетка сортирања алоцирати и користити кроз рекурзивне позиве.

Добијена функција сортирања има гарантовану сложеност најгорег случаја $O(n \log n)$, што значи да је много бржа од функција заснованих на сортирању селекцијом или сортирању уметањем чија је сложеност $O(n^2)$.

```
// ucesljava deo niza a iz intervala pozicija [i, m] i deo niza b iz
// intervala pozicija [j, n] koji su vec sortirani tako da se dobije
// sortiran rezultat koji se smesta u niz c, krenuvsi od pozicije k
void merge(vector<int>& a, int i, int m,
           vector<int>& b, int j, int n,
           vector<int>& c, int k) {
  while (i <= m && j <= n)
    c[k++] = a[i] <= b[j] ? a[i++] : b[j++];
  while (i <= m)
    c[k++] = a[i++];
  while (j <= n)
```

```
    c[k++] = b[j++];
}

// sortira deo niza a iz intervala pozicija [l, d] koristeci
// niz tmp kao pomocni
void merge_sort(vector<int>& a, int l, int d, vector<int>& tmp) {
  // ako je segment [l, d] jednoclan ili prazan, niz je vec sortiran
  if (l < d) {
    // sredina segmenta [l, d]
    int s = l + (d - l) / 2;
    // sortiramo segment [l, s]
    merge_sort(a, l, s, tmp);
    // sortiramo segment [s+1, d]
    merge_sort(a, s+1, d, tmp);

    // ucesljavamo segmente [l, s] i [s+1, d] smestajuci rezultat u
    // niz tmp
    merge(a, l, s, a, s+1, d, tmp, l);
    // vracamo rezultat iz niza tmp nazad u niz a
    for (int i = l; i <= d; i++)
      a[i] = tmp[i];

    // moze i pomocu biblioteckih funkcija
    /*
    merge(next(a.begin(), l), next(a.begin(), s+1),
          next(a.begin(), s+1), next(a.begin(), d+1),
          next(tmp.begin(), l));
    copy(next(tmp.begin(), l), next(tmp.begin(), d+1), next(a.begin(), l));
    */
  }
}

// sortira niz a
void merge_sort(vector<int>& a) {
  // alociramo pomocni niz
  vector<int> tmp(a.size());
  // pozivamo funkciju sortiranja
  merge_sort(a, 0, a.size() - 1, tmp);
}
```

Задатак: Број инверзија

Напиши програм који одређује колико у низу има инверзија (позиција $0 \leq i < j < n$, таквих да је $a_i > a_j$.

Улаз: Са стандардног улаза се уноси број n ($1 \leq n \leq 10^5$) и затим n целих бројева, сваки у посебном реду.

Излаз: На стандардни излаз исписати само тражени број инверзија.

Пример

Улаз	*Излаз*
5 3 1 4 2 5	3

Решење

Груба сила

Грубом силом се задатак решава тако што се помоћу угнежђених петљи испитају сви парови позиција $0 \leq i < j < n$ и преброје сви случајеви када је $a_i > a_j$ (бројимо елементе филтриране серије). Сложеност овог алгоритма одговара броју парова, а то је $O(n^2)$.

```
long long broj_inverzija(const vector<int>& a) {
  int n = a.size();
  long long broj = 0;
  for (int i = 0; i < n; i++)
    for (int j = i+1; j < n; j++)
      if (a[j] < a[i])
        broj++;
  return broj;
}
```

Подели па владај - модификација алгоритма MergeSort

Размотримо како бисмо проблем решили декомпозицијом. Празан и једночлан низ немају инверзија. Ако је низ подељен на две половине, укупан број инверзија једнак је збиру броја инверзија међу елементима прве половине, броја инверзија међу елементима друге половине и броја парова елемената где први елемент припада првој, други елемент припада другој половини и први је већи од другог. Прва два броја можемо одредити рекурзивно и остаје само питање како ефикасно одредити трећи број. Да бисмо добили укупну сложеност $O(n \log n)$ тај проблем је потребно решити у сложености $O(n)$ тако да испитивање свих парова елемената из прве и друге половине не долази у обзир. Задатак би се могао лакше решити ако би прва и друга половина биле сортиране (кључни увид је да сортирање елемената тих половина не мења трећи број). Тада можемо применити технику два показивача и веома слично као у случају обједињавања два сортирана низа одредити жељени трећи број. Уместо да сортирамо половине засебно, можемо алгоритам сортирања интегрисати са бројањем инверзија и проширити инваријанту наше функције (ојачати индуктивну хипотезу) тако да функција враћа број инверзија и уједно и сортира низ. На основу инваријанте, рекурзивни позиви ће сортирати леву и десну половину, а да бисмо је одржали, током одређивања трећег броја вршићемо обједињавање сортираних низова (исто као у алгоритму MergeSort).

Покажимо на једном примеру како можемо да пребројимо инверзије током обједињавања. Нека је дат низ $1, 3, 5, 4, 7, 6, 2, 8$. Све инверзије у њему су $(3, 2)$, $(5, 4)$, $(5, 2)$, $(4, 2)$, $(7, 6)$, $(7, 2)$ и $(6, 2)$ и има их 7.

Поделом се добијају половине $1, 3, 5, 4$ и $6, 7, 2, 8$. Рекурзивни позиви сортирају половине низа и у првој половини проналазе инверзију $(5, 4)$, а у другој половини инверзије $(7, 6)$, $(7, 2)$ и $(6, 2)$. Недостају још инверзије где је први елемент из прве, а други из друге половине низа (то су инверзије $(3, 2)$, $(4, 2)$ и $(5, 2)$). Сортиране половине су $1, 3, 4, 5$ и $2, 6, 7, 8$. Започнимо обједињавање ових низова.

- На почетку се пореде елементи 1 и 2. Пошто је елемент из леве половине мањи, он се пребацује у резултујући низ. Пошто је друга половина сортирана, знамо да је елемент 1 мањи од свих елемената друге половине, па он учествује у 0 инверзија.
- Након тога се пореде елементи 3 и 2. Овај пут је елемент десне половине 2 мањи од елемента из леве половине 3, па зато њега пребацујемо у резултат. Елемент 2 је мањи од елемента 3, а пошто је лева половина сортирана, мањи је и од свих елемената иза њега. Зато знамо да он учествује у 3 инверзије.
- Након тога се пореде елементи 3 и 6. Елемент 3 је мањи од елемента 6, па се он пребацује у резултат. Пошто се елемент 6 налази на позицији 1 у десној половини и пошто је десна половина сортирана, можемо закључити да елемент 3 учествује у једној инверзији (то је она са елементом 2).
- Након тога се пореде елементи 4 и 6. Елемент 4 је мањи од елемента 6, па се он пребацује у резултат. Пошто се елемент 6 налази на позицији 1 у десној половини и пошто је десна половина сортирана, можемо закључити да елемент 4 учествује у једној инверзији (то је она са елементом 2).
- Након тога се пореде елементи 5 и 6. Елемент 5 је мањи од елемента 6, па се он пребацује у резултат. Пошто се елемент 6 налази на позицији 1 у десној половини и пошто је десна половина сортирана, можемо закључити да елемент 5 учествује у једној инверзији (то је она са елементом 2).
- Пошто су сви елементи из прве половине преписани у резултујући низ, преписују се елементи из друге половине. Преписује се елемент 6 и пошто је он већи од свих елемената из прве половине, он не учествује ни у једној инверзији. Исто важи и за елементе 7 и 8.

Дакле, приликом обједињавања за сваки елемент можемо број инверзија у којима учествује (наравно, говоримо само о инверзијама између две половине низа).

- Када се у резултујући низ преписује елемент из леве половине низа он је строго већи од свих елемената десне половине који су већ преписани (њихов број се лако одреди на основу вредности десног показивача), а мањи или једнак од осталих елемената десне половине, па је број инверзија у којима он учествује једнак вредности десног показивача (од које треба одузети позицију почетка десне половине низа, ако њени елементи нису смештени од позиције 0). Ако десна половина почиње на позицији $s + 1$, тада се број инверзија може одредити као $p_d - s - 1$.

- Када се у резултујући низ преписује елемент из десне половине низа, он је строго мањи од текућег елемента у левој половини и свих елемената иза њега. Дакле, број инверзија у којима учествује може се израчунати тако што се од укупног броја елемената леве половине одузме позиција левог показивача (наравно, опет је потребно у обзир узети и позицију на којој почиње лева половина низа). Ако се лева половина завршава на позицији s онда се број инверзија може израчунати као $s + 1 - p_l$.

Укупан број инверзија се може израчунати било тако што се сабере број појединачних инверзија за сваки елемент из леве половине (тада бројач увећавамо у тренутку када се пребацује елемент из леве половине) или тако што се сабере број појединачних инверзија за сваки елемент из десне половине (тада бројач увећавамо у тренутку када се пребацује елемент из десне половине). Можда је мало једноставније сабрати инверзије свих елемената у десној половини, јер тада у фази преписивања елемената преостале половине (када се једна половина исцрпи) нема потребе за ажурирањем броја инверзија. Заиста, ако су преостали само елементи леве половине, за сваки елемент десне половине је већ израчунат и сабран број инверзија, а ако су препостали само елементи десне половине, пошто су исцрпљени сви елементи леве половине, ти преостали елементи десне половине не учествују ни у једној инверзији. Ипак, пошто рекурзивна функција поред израчунавања броја инверзија треба да сортира низ, преостале елементе морамо преписати у резултат (чак иако број инверзија знамо и раније).

```
long long broj_inverzija(vector<int>& a, int l, int d, vector<int>& b) {
  if (l >= d)
    return 0;
  int s = l + (d - l) / 2;
  long long broj = 0;
  broj += broj_inverzija(a, l, s, b);
  broj += broj_inverzija(a, s+1, d, b);
  int pl = l, pd = s+1, pb = 0;
  while (pl <= s && pd <= d) {
    if (a[pl] <= a[pd])
      b[pb++] = a[pl++];
    else {
      broj += s - pl + 1;
      b[pb++] = a[pd++];
    }
  }
  while (pl <= s)
    b[pb++] = a[pl++];
  while (pd <= d)
    b[pb++] = a[pd++];

  copy(begin(b), next(begin(b), d - l + 1), next(begin(a), l));

  return broj;
}

long long broj_inverzija(const vector<int>& a) {
  vector<int> tmp1(a.size()), tmp2(a.size());
  copy(begin(a), end(a), begin(tmp1));
  return broj_inverzija(tmp1, 0, a.size()-1, tmp2);
}
```

Задатак: Силуета града

Са брода се виде зграде на обали велеграда. Дуж обале је постављена координатна оса и за сваку зграду се зна позиција левог краја, висина и позиција десног краја. Написати програм који израчунава силуету града.

Силуета је део-по-део константна функција и одређена је интервалима константности $(-\infty, x_0)$, $[x_0, x_1)$, $[x_1, x_2)$, ..., $[x_{n-1}, +\infty)$, одређеним тачкама поделе $x_0 < x_1 < \ldots < x_{n-1}$ и вредностима 0, h_0, ..., h_{n-2} и h функције на сваком од интервала.

$$\begin{array}{cccccccccccc} & 0 & & h_0 & & h_1 & & \ldots & & h_{n-2} & & 0 & \\ -\infty & & x_0 & & x_1 & & x_2 & \ldots & x_{n-2} & & x_{n-1} & & +\infty \end{array}$$

Подразумевамо да су крајње тачке $-\infty$ и $+\infty$ и да су вредности на тим интервалима једнаке нули. Дакле, део-по-део константна функција се може представити помоћу n тачака $x_0, \ldots x_{n-1}$ и $n-1$ вредности $h_0, \ldots, h_{n-2}$. Једноставности ради ми ћемо овакве функције представљати помоћу n уређених парова (x_0, h_0), (x_1, h_1), ..., (x_{n-2}, h_{n-2}) и $(x_{n-1}, 0)$. Дакле, наш алгоритам прима низ уређених тројки који описује појединачне зграде, а враћа низ уређених парова који описује силуету.

Улаз: Са станадардног улаза се учитава број зграда n, а затим, у n наредних линија по три цела броја раздвојена са по једним размаком: леви крај зграде, десни крај зграде и висина зграде.

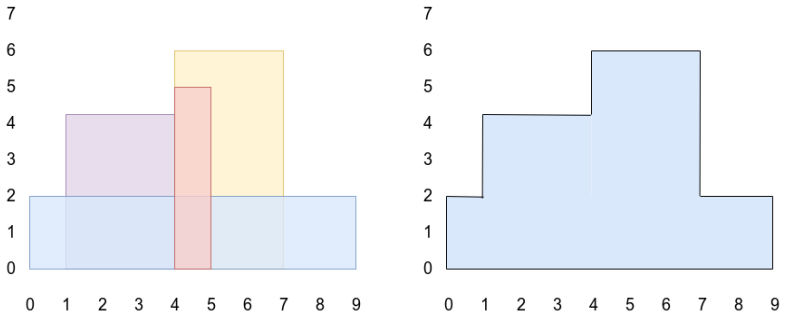
Излаз: На стандардни излаз исписати силуету описану преко низа промена висина.

Пример

Улаз	*Излаз*
`4`	`0 2`
`1 4 4`	`1 4`
`4 5 5`	`4 6`
`4 7 6`	`7 2`
`0 9 2`	`9 0`

Објашњење

Са леве стране су приказане зграде, а са десне силуета.



Слика 5.1: Зграде и силуета

Решење

Основна индуктивна конструкција подразумева да умемо да нађемо силуету све осим последње зграде и да на крају умемо да ту зграду уклопимо у силуету осталих зграда. Дакле, обилазимо зграду по зграду и једну по једну убацујемо у већ одређену силуету претходних зграда.

Да бисмо зграду уклопили у постојећу силуету потребно нам је линеарно време. Наиме, иако се део силуете у који се зграда интегрише може пронаћи бинарном претрагом (то је део силуете између левог и десног краја зграде), ако се елементи чувају у низу, уметање нових елемената у низ и евентуално брисање постојећих елемената захтева линеарно време. Ако промене чували у повезаној листи уместо у низу, тада бисмо уметање нових промена и брисање промена из силуете могли вршити у константном времену, међутим, не бисмо могли користити бинарну претрагу. У сваком случају, потребно је да прођемо и пронализирамо све елементе постојеће силуете у делу где се умеће нова зграда (између њеног левог и десног краја), а њих може да буде

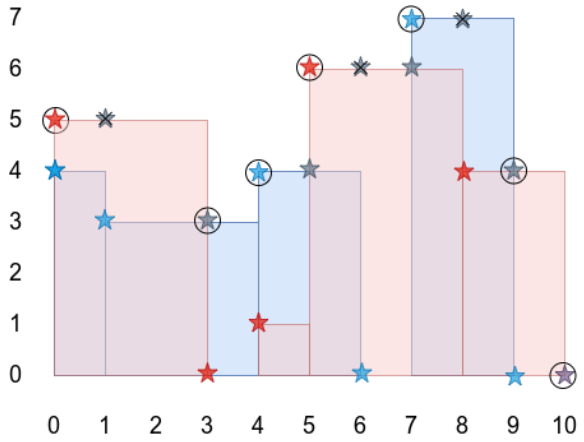
пуно ако је зграда широка и њихова обрада захтева време (n), где је n текући број елемената силуете. То значи да ће решење бити сложености $T(n) = T(n-1) + O(n), T(1) = O(1)$, што даје сложеност $O(n^2)$, где је n број зграда.

Подели па владај

Проблем можемо ефикасно решити техником подели-па-владај, веома слично алгоритму сортирања обједињавањем (енгл. merge sort). Кључна опаска је то да две силуете можемо објединити за исто време за које можемо објединити једну зграду у силуету (уз претпоставку да ће нова силуета бити смештена у посебном низу). Пошто су резултујуће силуете сортиране можемо их обједињавати веома слично обједињавању два сортирана низа бројева. Тај је алгоритам описан у задатку Обједињавање.

Током рада алгоритма одржаваћемо два индекса (показивача) који одређују текуће елементе у левој тј. у десној силуети. Одржаваћемо и две променљиве и у којима се чува тренутна висина леве тј. десне силуете. Ако је неки од показивача стигао до краја, само ћемо преписивати преостале елементе из силуете која још није потпуно обрађена. У супротном ћемо поредити x координате текућих елемената обе силуете. Ако је x координата текућег елемента у левој силуети мања, ажурираћемо текућу висину леве силуете и увећати леви показивач, ако је x координата текућег елемента у десној силуети мања, ажурираћемо висину десне силуете и увећати десни показивач, а ако су x координате текућих елемената обе силуете једнаке, ажурираћемо висине обе зграде и увећаваћемо оба показивача. Након тога ћемо у резултујућу силуету додавати x координату управо одабраног елемента и придружићемо јој већу од текућих висина две силуете. При том ћемо водити рачуна да у резултат не додајемо нови елемент ако је његова висина једнака висини претходног елемента у резултујућој силуети.

Прикажимо како се врши обједињавање две силуете на једном примеру.



Слика 5.2: Пример две силуете које се обједињавају

Обједињавамо силуете: $L : (0,5), (3,0), (4,1), (5,6), (8,4), (10,0)$ и $D : (0,4), (1,3), (4,4), (6,0), (7,7), (9,0)$.

	x	Hl	Hd	H	dodaje se	silueta
		0	0			
(0,5), (0,4)	0	5	4	5	(0,5)	(0,5)
(1,3)	1	5	3	5	(1,5)	(0,5)
(3,0)	3	0	3	3	(3,3)	(0,5),(3,3)
(4,1), (4,4)	4	1	4	4	(4,4)	(0,5),(3,3),(4,4)
(5,6)	5	6	4	6	(5,6)	(0,5),(3,3),(4,4),(5,6)
(6,0)	6	6	0	6	(6,6)	(0,5),(3,3),(4,4),(5,6)
(7,7)	7	6	7	7	(7,7)	(0,5),(3,3),(4,4),(5,6),(7,7)
(8,4)	8	4	7	7	(8,7)	(0,5),(3,3),(4,4),(5,6),(7,7)
(9,0)	9	4	0	4	(9,4)	(0,5),(3,3),(4,4),(5,6),(7,7),(9,4)
(10,0)	10	0	0	0	(10,0)	(0,5),(3,3),(4,4),(5,6),(7,7),(9,4),(10,0)

Пошто се обједињавање две силуете може урадити у линеарном времену у односу на укупан број зграда у обе силуете, укупна сложеност се израчунава из једначине $T(n) = 2T(n/2) + O(n), T(1) = O(1)$, чије је решење $O(n \log n)$.

Напоменимо да би се имплементација могла убрзати ако би се за обједињавање користио један помоћни вектор (у тренутној имплементацији се одређено време губи на динамичку алокацију елемената резултујућег вектора).

```
// zgrada je odredjena pocetkom a, krajem b i visinom h
struct zgrada {
  int a, b, h;
  zgrada(int a = 0, int b = 0, int h = 0)
    : a(a), b(b), h(h) {
  }
};

// silueta je odredjena nizom promena
// svaka promena je odredjena koordinatom x i visinom h
struct promena {
  int x, h;
  promena(int x = 0, int h = 0)
    : x(x), h(h) {
  }
};


// integrisemo promenu (x, h) u postojecu siluetu
void dodajPromenu(vector<promena>& silueta, int x, int h) {
  // ako je siuleta prazna ili ako je prethodna visina razlicita od
  // tekuce, dodajemo novu promenu u niz
  if (silueta.empty() || h != silueta.back().h)
      // dodajemo novu promenu u niz
      silueta.emplace_back(x, h);
}

// odredjuje se silueta zgrada na pozicijama [l, d]
vector<promena> silueta(const vector<zgrada>& zgrade, int l, int d) {
  vector<promena> rezultat;

  // silueta koja odgovara jednoj zgradi
  if (l == d) {
    rezultat.emplace_back(zgrade[l].a, zgrade[l].h);
    rezultat.emplace_back(zgrade[l].b, 0);
    return rezultat;
  }

  // određujemo posebno siluete za prvu i drugu polovinu zgrada
  int s = l + (d - l) / 2;
  vector<promena> rezultat_l = silueta(zgrade, l, s);
  vector<promena> rezultat_d = silueta(zgrade, s+1, d);

  // objedinjujemo dve siluete

  // tekući indeksi i visine u levoj i desnoj silueti
  int ll = 0, dd = 0;
  int Hl = 0, Hd = 0;
  // dok god postoji neka neobrađena promena
  while (ll < rezultat_l.size() || dd < rezultat_d.size()) {
    // određujemo novu tačku promene
    int x;
    // ako smo završili sa levom siluetom samo prebacujemo zgrade iz desne
    if (ll == rezultat_l.size()) {
      x = rezultat_d[dd].x; Hd = rezultat_d[dd].h;
```

```
      dd++;
    // ako smo završili sa desnom siluetom samo prebacujemo zgrade iz desne
    } else if (dd == rezultat_d.size()) {
      x = rezultat_l[ll].x; Hl = rezultat_l[ll].h;
      ll++;
    } else {
      // određujemo raniju od tekućih promena leve i desne siluete
      int xl = rezultat_l[ll].x;
      int xd = rezultat_d[dd].x;
      if (xl < xd) {
        x = xl; Hl = rezultat_l[ll].h;
        ll++;
      } else if (xl > xd) {
        x = xd; Hd = rezultat_d[dd].h;
        dd++;
      } else {
        x = xl; Hl = rezultat_l[ll].h; Hd = rezultat_d[dd].h;
        ll++; dd++;
      }
    }

    // veća od dve tekuće visine
    int h = max(Hl, Hd);

    // integrišemo promenu (x, h) u tekuću rezultujuću siluetu
    dodajPromenu(rezultat, x, h);
  }

  return rezultat;
}

// odredjuje se silueta niza zgrada
vector<promena> silueta(const vector<zgrada>& zgrade) {
   return silueta(zgrade, 0, zgrade.size() - 1);
}
```

Задатак: Максимални збир сегмента

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. *Види текст задатка.*

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Разлагање на потпроблеме

Један начин да решимо овај проблем је заснован на техници разлагања. Декомпозиција нам сугерише да је пожељно да низ поделимо на два подниза једнаке дужине чија решења можемо да конструишемо на основу индуктивне хипотезе (најчешће рекурзивним позивима). Базу и овај пут чини случај празног низа, који садржи само празан сегмент чији је збир нула. Фиксирајмо средишњи елемент низа. Све сегменте низа можемо да групишемо у три групе: сегменте који су у потпуности лево од средишњег елемента, сегменте који су у потпуности десно од средишњег елемента и сегменте који садрже средишњи елемент. Највеће збирове сегмената у првој и у другој групи знамо на основу индуктивне хипотезе. Највећи збир сегмента у трећој групи можемо лако одредити анализом свих сегмената: крећемо од једночланог сегмента који садржи само средишњи елемент и инкрементално се ширимо налево додајући један по један елемент и рачунајући текући максимум, а затим крећемо од максималног сегмента проширеног налево и инкрементално га проширујемо једним по једним елементом надесно, тражећи нови максимум.

Ако са n означимо дужину низа $d - l + 1$ и ако време извршавања обележимо са $T(n)$, тада важи да је

$T(0) = O(1)$ и да је $T(n) = 2T(n/2) + O(n)$. Наиме, врше се два рекурзивна позива за дупло мање низове, а највећи збир сегмената који обухватају средишњи елемент израчунавамо у времену $O(n)$ (што је прилично очигледно јер имамо две петље које се укупно извршавају n пута, а чија су тела константне сложености). На основу мастер теореме лако се закључује да је $T(n) = O(n \log n)$. Дакле, овај алгоритам је мање ефикасан алгоритама сложености $O(n)$, али је и даље прилично употребљив (и сигурно је много бољи од алгоритама грубе силе који су квадратне или кубне сложености).

```
int maksZbirSegmenta(const vector<int>& a, int l, int d) {
  if (l > d)
    return 0;
  int s = l + (d - l) / 2;
  int maks_zbir_levo = maksZbirSegmenta(a, l, s-1);
  int maks_zbir_desno = maksZbirSegmenta(a, s+1, d);
  int zbir_sredina = a[s];
  int maks_zbir_sredina = zbir_sredina;
  for (int i = s-1; i >= l; i--) {
    zbir_sredina += a[i];
    if (zbir_sredina > maks_zbir_sredina)
      maks_zbir_sredina = zbir_sredina;
  }
  zbir_sredina = maks_zbir_sredina;
  for (int i = s+1; i <= d; i++) {
    zbir_sredina += a[i];
    if (zbir_sredina > maks_zbir_sredina)
      maks_zbir_sredina = zbir_sredina;
  }
  return max({maks_zbir_levo, maks_zbir_desno, maks_zbir_sredina});
}

int maksZbirSegmenta(const vector<int>& a) {
  return maksZbirSegmenta(a, 0, a.size() - 1);
}
```

Ојачање индуктивне хипотезе

На идеји декомпозиције можемо изградити и ефикаснији алгоритам. Кључни увид је да се највећи збир сегмента око средњег елемента може добити као збир највећег суфикса низа лево од тог елемента и највећег префикса низа десно од тог елемента. Можемо ојачати индуктивну хипотезу и уместо да префикс и суфикс рачунамо у петљи, у линеарном времену, можемо претпоставити да за обе половине низа префикс и суфикс добијамо као резултат рекурзивног позива. То нам је довољно да одредимо максимални збир функције, али морамо „вратити дуг'' и наша функција сада поред максималног збира сегмента мора израчунати и максимални збир префикса и максимални збир суфикса целог низа. Максимални збир префикса целог низа је већи број од максималног збира префикса левог дела и од збира целог левог дела и максималног збира префикса десног дела. Слично, максимални збир суфикса целог низа је већи од максималног збира суфикса десног дела и од збира максималног збира суфикса левог дела и целог десног дела. Зато је неопходно додатно ојачати индуктивну хипотезу и током рекурзије рачунати и збир целог низа.

Претпоставимо да је P збир максималног префикса низа, да је S збир максималног суфикса низа, да је Z збир целог низа и да је M максимални збир сегмента низа. Означимо индексом l те статистике у левој половини низа и индексом d те статистке у десној половини низа. Тада важи:

$$\begin{aligned} Z &= Z_l + Z_d \\ P &= \max(P_l, Z_l + P_d) \\ S &= \max(S_d, S_l + Z_d) \\ M &= \max(M_l, M_d, S_l + P_d) \end{aligned}$$

Једначина која описује ову рекурзију је $T(n) = 2T(n/2) + O(1)$, па је сложеност овог решења линеарна тј. $O(n)$.

```
void maksZbirSegmenta(const vector<int>& a, int l, int d,
                      int& zbir, int& maks_zbir,
                      int& maks_prefiks, int& maks_sufiks) {
  if (l == d) {
    zbir = maks_zbir = maks_prefiks = maks_sufiks = a[l];
    return;
  }
  int s = l + (d - l) / 2;
  int zbir_levo, maks_zbir_levo, maks_sufiks_levo, maks_prefiks_levo;
  maksZbirSegmenta(a, l, s,
                   zbir_levo, maks_zbir_levo,
                   maks_prefiks_levo, maks_sufiks_levo);
  int zbir_desno, maks_zbir_desno, maks_sufiks_desno, maks_prefiks_desno;
  maksZbirSegmenta(a, s+1, d,
                   zbir_desno, maks_zbir_desno,
                   maks_prefiks_desno, maks_sufiks_desno);
  zbir = zbir_levo + zbir_desno;
  maks_prefiks = max(maks_prefiks_levo, zbir_levo + maks_prefiks_desno);
  maks_sufiks = max(maks_sufiks_desno, maks_sufiks_levo + zbir_desno);
  maks_zbir = max({maks_zbir_levo, maks_zbir_desno,
                   maks_sufiks_levo + maks_prefiks_desno});
}

int maksZbirSegmenta(const vector<int>& a) {
  int zbir, maks_zbir, maks_prefiks, maks_sufiks;
  maksZbirSegmenta(a, 0, a.size() - 1,
                   zbir, maks_zbir, maks_prefiks, maks_sufiks);
  return maks_zbir;
}
```

Задатак: Најближи пар тачака

У датом скупу тачака у равни одредити колико је растојање између две тачке које су међусобно најближе.

Улаз: Са стандардног улаза се уноси број тачака n ($1 \leq n \leq 50000$), а затим у наредних n редова координате тачака (два цела броја између -10^9 и 10^9, раздвојена размаком).

Излаз: На стандардни излаз исписати тражено растојање, заокружено на пет децимала.

Пример

Улаз	*Излаз*
5 0 0 0 2 2 0 2 2 1 1	1.41421

Решење

Груба сила

Решење грубом силом подразумева испитивање свих парова тачака и сложеност му је $O(n^2)$.

```
struct Tacka {
  int x, y;
};

double rastojanje(const Tacka& t1, const Tacka& t2) {
  double dx = t1.x - t2.x;
  double dy = t1.y - t2.y;
```

```
  return sqrt(dx*dx + dy*dy);
}

double najblizeTacke(vector<Tacka>& tacke) {
  int n = tacke.size();
  double d = numeric_limits<double>::max();
  for (int i = 0; i < n; i++)
    for (int j = i+1; j < n; j++)
      d = min(d, rastojanje(tacke[i], tacke[j]));
  return d;
}
```

Декомпозиција

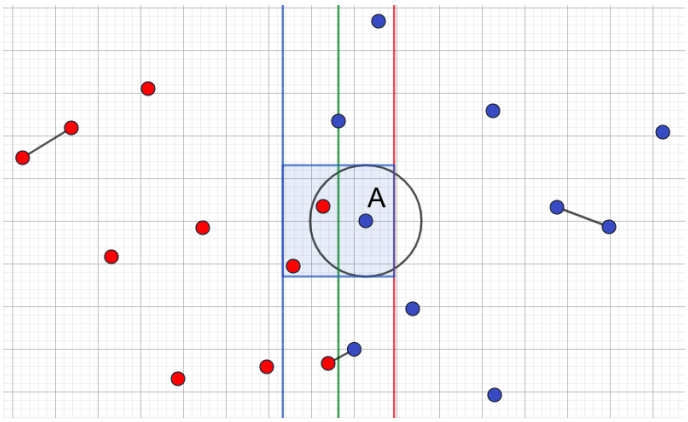
Један начин да се до решења дође ефикасније је да се примени декомпозиција.

Базни случај представља ситуација у којој имамо мање од четири тачке, јер њих не можемо поделити у две половине у којима постоји бар по један пар тачака (а ако у скупу немамо бар 2 тачке, најмање растојање није јасно дефинисано). У том случају решење налазимо поређењем растојања свих парова тачака (пошто је тачака мало, овај корак је сложености $O(1)$).

Скуп тачака можемо једном вертикалном линијом поделити на две отприлике истобројне половине. Ако тачке сортирамо по координати x, вертикална линија може одговарати координати средишње тачке. Рекурзивно одређујемо најмање растојање у првој половини (те тачке се налазе лево од вертикалне линије или евентуално на њој) и у другој половини (те тачке се налазе десно од вертикалне линије или евентуално на њој). Најближи пар је такав да су (1) обе тачке у левој половини, (2) обе тачке у десној половини или (3) једна тачка је у левој, а друга у десној половини. За прва два случаја већ знамо решења (на основу резултата рекурзивних позива) и остаје да се размотри само трећи.

Нека је d_l минимално растојање тачака у левој половини, d_r минимално растојање тачака у десној половини, а d мање од та два растојања. Ако вертикална линија има x-координату x, тада је могуће одбацити све тачке које су лево од $x - d$ и десно од $x + d$, јер је њихово растојање до најближе тачке из супротне половине сигурно веће од d. Потребно је испитати све преостале тачке, тј. све тачке из појаса $[x - d, x + d]$, проверити да ли међу њима постоји неки пар тачака чије је растојање строго мање од d и вредност d ажурирати на вредност најмањег растојања таквог пара тачака. Ако су тачке насумично распоређене, реално је очекивати да ће већина тачака бити ван тог појаса. Међутим, проблем је то што у најгорем случају у појасу може бити пуно тачака (могуће је чак и да се свих n тачака нађе у том појасу) и ако испитујемо све парове, долазимо у најгорем случају до око $n^2/4$ поређења (ако је пола тачака лево, а пола десно од линије поделе). Ипак, проверу је могуће организовати тако да се провери само мали број парова тачака.

Једноставности ради ћемо претпоставити да на исти начин разматрамо све тачке унутар појаса $[x-d, x+d]$, без обзира са које стране вертикалне линије се налазе (унапред знамо да је провера тачака које су са исте стране вертикалне линије поделе непотребна, али не може нарушити коректност, док год смо сигурни да се пореде и сви потребни парови тачака са различите стране те линије). Сваку тачку A из појаса је довољно упоредити само са оним тачкама које леже унутар круга са центром у тачки A и полупречником d, јер су све тачке ван тог круга сигурно од тачке A удаљене више од d, што омогућава значајна одсецања. Међутим, припадност кругу није једноставно проверити и зато уместо њега можемо разматрати квадрат странице дужине $2d$, чији се центар налази на правој $y = x$, а на чијој се хоризонталној средњој линији налази тачка A и тачку A ћемо поредити само са тачкама унутар тог квадрата (знамо да су тачке ван тог квадрата сигурно на растојању већем од d). Тиме ће одсецање бити за нијансу мање него у случају круга (јер су неке тачке унутар квадрата на растојању већем од d), али ће детектовање тачака које припадају том квадрату бити веома једноставно – то ће бити све оне тачке из појаса $[x - d, x + d]$, којима је координата y у интервалу $[y_A - d, y_A + d]$.



Слика 5.3: Најближи пар тачака у левом појасу, десном појасу и између појасева. Круг и квадрат који садрже тачке које се пореде са тачком A.

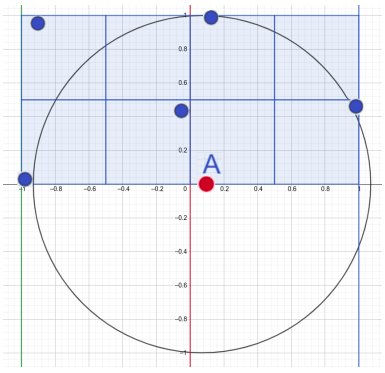
Додатно смањење броја поређења можемо добити ако приметимо да сваки пар обрађујемо два пута (једном док обрађујемо тачке у околини прве, а једном док обрађујемо тачке у околини друге тачке). Можемо једноставно закључити да је довољно сваку тачку A поредити само са оним тачкама које се су изнад ње (или су евентуално на истој висини као она), тј. не у целом квадрату, него само у његовој горњој половини. Дакле, сваку тачку A је потребно упоредити само са тачкама чије x координате леже унутар интервала $[x - d, x + d]$ и чије y координате леже унутар интервала $[y_A, y_A + d]$. Први услов можемо обезбедити тако што пре поређења све тачке из појаса ширине d око вертикалне линије поделе издвојимо у посебан низ (за то нам је потребно $O(n)$ додатне меморије и времена). Други услов ефикасније можемо обезбедити ако све тачке тог помоћног низа сортирамо по координати y (за то нам је потребно време $O(n \log n)$ и затим тачке обрађујемо у неопадајућем редоследу y координата. За сваку тачку A обрађујемо само тачке које се налазе након ње у сортираном низу и обрађујемо једну по једну тачку све док не наиђемо на тачку чија је координата y већа или једнака од вредности $y_A + d$ (она од тачке A не може бити на мањем растојању од d, а исто важи и за све тачке у низу иза ње).

```
// funkcija pronalazi najblizi par tacaka u delu nizu [l, r]
double najblizeTacke(vector<Tacka>& tacke, int l, int r, vector<Tacka>& pojas) {
  // za manje od 4 tacke, grubom silom odredjujemo najblizi par
  if (r - l + 1 < 4) {
    double d = numeric_limits<double>::max();
    for (int i = l; i < r; i++)
      for (int j = i+1; j <= r; j++)
        d = min(d, rastojanje(tacke[i], tacke[j]));
    return d;
  }

  // delimo niz tacaka sa pozicija [l, r] u dve polovine
  int s = l + (r - l) / 2;

  // rekurzivno pronalazimo najblizi par u levoj i desnoj polovini niza
  double d1 = najblizeTacke(tacke, l, s, pojas);
  double d2 = najblizeTacke(tacke, s+1, r, pojas);

  // najmanje rastojanje svih parova tacaka
  double d = min(d1, d2);

  // pronalazimo tacke u pojasu sirine 2d oko sredisnje linije
  double dl = tacke[s].x - d, dr = tacke[s].x + d;
  int k = 0;
  for (int i = l; i <= r; i++)
    if (dl <= tacke[i].x && tacke[i].x <= dr)
      pojas[k++] = tacke[i];
```

```
  // sortiramo tacke u pojasu po y koordinati
  sort(begin(pojas), next(begin(pojas), k),
       [](const Tacka& t1, const Tacka& t2) {
         return t1.y < t2.y;
       });

  // analiziramo sve tacke u pojasu
  for (int i = 0; i < k; i++)
    // svaku tacku poredimo samo sa onim tackama koje su u
    // pravougaoniku iznad nje
    for (int j = i+1; j < k && pojas[j].y - pojas[i].y < d; j++)
      d = min(d, rastojanje(pojas[i], pojas[j]));

  // vracamo najkrace rastojanje
  return d;
}

double najblizeTacke(vector<Tacka>& tacke) {
  // sortiramo tacke po x koordinati
  sort(begin(tacke), end(tacke),
       [](const Tacka& t1, const Tacka& t2) {
         return t1.x < t2.x;
       });
  // pomocni niz (alociramo ga samo jednom, van rekurzije)
  vector<Tacka> pojas(tacke.size());
  return najblizeTacke(tacke, 0, tacke.size() - 1, pojas);
}
```

Одредимо сложеност претходног алгоритма. Алгоритам се састоји од два рекурзивна позива за двоструко мању димензију низа тачака и фазе добијања крајњег резултата на основу резултата рекурзивних позива и додатне анализе тачака у појасу $[x - d, x + d]$. Већ смо констатовали да издвајање тачака централног појаса захтева $O(n)$ меморије и времена и да сортирање тих тачака по координати y захтева додатних $O(n \log n)$ корака. Остаје још да се процени сложеност угнежђених петљи у којима се пореде тачке унутар појаса. Иако делује да је сложеност квадратна, елементарним геометријским резоновањем доказаћемо да је сложеност тог корака линеарна тј. $O(n)$ и да се у сваком кораку спољашње петље унутрашња петља може извршити само веома мали број пута (доказаћемо да је тај број извршавања ограничен одозго са 7, мада је у пракси он често и доста мањи од тога и за насумично генерисане тачке та петља се најчешће извршава 0, 1 или евентуално 2 пута).

За сваку тачку A можемо конструисати 8 квадрата димензије $d/2$, као што је приказано на слици (квадрати су уписани у појас $[x - d, x + d]$, у два реда од по четири квадрата и тачка A лежи на доњој ивици доњих квадрата).



Слика 5.4: Најближи пар тачака

Највеће растојање између две тачке унутар неког квадрата се постиже када они леже у његовим наспрамним теменима, а пошто је дужина дијагонале квадрата странице $\frac{d}{2}$ једнака $\frac{d\sqrt{2}}{2} \approx 0,70711 \cdot d$, растојање између сваке две тачке унутар истог квадрата је строго мање од d. Пошто сви квадрати леже било потпуно са леве стране вертикалне линије поделе, било са њене десне стране унутар сваког од квадрата се може наћи највише једна тачка нашег скупа (у супротном би се било са леве, било са десне стране централне линије поделе налазио пар тачака са растојањем строго мањим од d, што је контрадикторно са дефиницијом величине d). То значи да се изнад тачке A може налазити највише 7 тачака које припадају осталим квадратима (сама тачка A већ припада једном од квадрата) и да се све остале тачке које су изнад A налазе и изнад наших квадрата, што значи да им је растојање од A сигурно веће од d (јер им је вертикално растојање веће од d) и њих није потребно разматрати.

Тачке које су са исте стране линије поделе као и тачка A можемо просто прескочити у телу унутрашње петље и тако уштедити на рачунању њиховог растојања од тачке A, али експерименти показују да та уштеда није осетна. Друга могућност за имплементацију је да не чувамо све тачке из појаса у истом скупу, већ да их поделимо у два појаса и да затим да обрадимо прво све тачке из левог појаса гледајући растојања у односу на наредне највише 4 тачке из десног појаса, а затим да обрадимо све тачке из десног појаса гледајући растојања у односу на највише 4 тачке из левог појаса (јер у супротном појасу постоји 4 квадрата димезије $d/2$, за које смо доказали да не могу да садрже две тачке истовремено). Имплементација на тај начин је мало компликованија, а експерименти не указују на значајне добитке.

Дакле, након рекурзивних позива, за добијање коначног резултата је потребно извршити додатних $O(n \log n)$ корака и декомпозиција задовољава рекурентну једначину $T(n) = 2T(n/2) + O(n \log n)$. Решење ове једначине, на основу мастер теореме, је $O(n(\log n)^2)$.

Сортирање обједињавањем

Сложеност се може поправити ако се сортирање по координати y врши истовремено са проналажењем најближег пара тачака, тј. ако се ојача индуктивна хипотеза и ако се претпостави да ће рекурзивни позив вратити растојање између најближе две тачке и уједно сортирати дате тачке по координати y. У кораку обједињавања два сортирана низа обједињујемо у један. То можемо урадити уобичајеним алгоритмом обједињавања, заснованом на техници два показивача, који обједињавање врши у линеарној сложености. У језику C++ тај алгоритам је доступан и помоћу библиотечке функције `merge`. На тај начин добијамо алгоритам који задовољава једначину $T(n) = 2T(n/2) + O(n)$ и сложености је $O(n \log n)$. Нагласимо да ова оптимизација није револуционарна, али може мало побољшати ефикасност.

На нивоу имплементације, мало побољшање бисмо могли добити и тако што бисмо избегли алокације помоћног вектора унутар рекурзивних позива и код изменити тако да се у сваком рекурзивном позиву користи исти, унапред алоциран помоћни вектор. Још једна могућа оптимизација о којој би се могло размислити је смањивање броја операција кореновања.

```
bool porediX(const Tacka& t1, const Tacka& t2) {
  return t1.x < t2.x;
}

bool porediY(const Tacka& t1, const Tacka& t2) {
  return t1.y < t2.y;
}

// funkcija pronalazi najblizi par tacaka u delu nizu [l, r] i dodatno
// sortira tacke unutar tog dela niza po y koordinati
double najblizeTacke(vector<Tacka>& tacke, int l, int r, vector<Tacka>& pojas) {
  // za manje od 4 tacke, grubom silom odredjujemo najblizi par
  if (r - l + 1 < 4) {
    // odredjujemo najblizi par
    double d = numeric_limits<double>::max();
    for (int i = l; i < r; i++)
      for (int j = i+1; j <= r; j++)
        d = min(d, rastojanje(tacke[i], tacke[j]));
    // sortiramo tacke
    sort(next(begin(tacke), l), next(begin(tacke), r+1), porediY);
```

```
    return d;
  }

  // delimo niz tacaka sa pozicija [l, r] u dve polovine
  int s = l + (r - l) / 2;
  int x = tacke[s].x;
  // rekurzivno pronalazimo najblizi par u levoj i desnoj polovini niza
  // sortirajuci te polovine po y koordinati
  double d1 = najblizeTacke(tacke, l, s, pojas);
  double d2 = najblizeTacke(tacke, s+1, r, pojas);

  // najmanje rastojanje svih parova tacaka
  double d = min(d1, d2);

  // objedinjavamo dva sortirane polovine (koristimo niz pojas kao pomocni)
  merge(next(begin(tacke), l), next(begin(tacke), s+1),
        next(begin(tacke), s+1), next(begin(tacke), r+1),
        begin(pojas), porediY);
  copy(begin(pojas), next(begin(pojas), r - l + 1), next(begin(tacke), l));

  // pronalazimo tacke u pojasu sirine 2d oko sredisnje linije
  double dl = x - d, dr = x + d;
  for (int i = l; i <= r; i++)
    if (dl <= tacke[i].x && tacke[i].x <= dr)
      pojas[k++] = tacke[i];

  // analiziramo sve tacke u pojasu
  for (int i = 0; i < k; i++)
    // svaku tacku poredimo samo sa onim tackama koje su u
    // pravougaoniku iznad nje
    for (int j = i+1; j < k && pojas[j].y - pojas[i].y < d; j++)
      d = min(d, rastojanje(pojas[i], pojas[j]));

  return d;
}

double najblizeTacke(vector<Tacka>& tacke) {
  sort(begin(tacke), end(tacke), porediX);
  vector<Tacka> pojas(tacke.size());
  return najblizeTacke(tacke, 0, tacke.size() - 1, pojas);
}
```

Задатак: Множење полинома

Напиши програм који ефикасно одређује производ два полинома.

Улаз: Са стандардног улаза се носе два полинома. За сваки полином је у једној линији дат степен n (цео број између 1 и 50000), а затим у наредној линији коефицијенти (реални бројеви заокружени на једну децималу, раздвојени размацима). Коефицијенти се задају редом, кренувши од слободног члана тј. коефицијента уз x^0, па закључно са коефицијентом уз x^n.

Излаз: На стандардни излаз исписати полином производ, у истом формату у ком су задати и чиниоци, осим што сви коефицијенти треба да буду заокружени на две децимале.

Пример

Улаз	*Излаз*
2 1.0 2.0 3.0 1 1.0 2.0	3 1.00 4.00 7.00 6.00

Решење

Множење можемо извршити класичним алгоритмом множења, који смо приказали у задатку Аритметика над полиномима. Сложеност тог алгоритма је $O(n_1 \cdot n_2)$, где су n_1 и n_2 степени полинома који се множе.

```
// funkcija mnozi dva polinoma p1*p2
vector<double> proizvod(const vector<double>& p1,
                        const vector<double>& p2) {
  int n1 = p1.size(), n2 = p2.size();
  vector<double> proizvod(n1+n2-1, 0);
  for (int i = 0; i < n1; i++)
    for (int j = 0; j < n2; j++)
      proizvod[i+j] += p1[i] * p2[j];
  return proizvod;
}

vector<double> ucitajPolinom() {
  int n;
  cin >> n;
  vector<double> p(n+1, 0.0);
  for (int i = 0; i <= n; i++)
    cin >> p[i];
  return p;
}

int main() {
  vector<double> p1 = ucitajPolinom();
  vector<double> p2 = ucitajPolinom();
  vector<double> r = proizvod(p1, p2);
  cout << r.size() - 1 << endl;
  for (size_t i = 0; i < r.size(); i++)
    cout << fixed << showpoint << setprecision(2) << r[i] << " ";
  cout << endl;
  return 0;
}
```

Иако се неко време сматрало да је доња граница сложености множења полинома $O(n_1 \cdot n_2)$, Анатолиј Карацуба је 1960. показао да је декомпозицијом могуће добити ефикаснији алгоритам. Претпоставимо да је потребно помножити полиноме $a + bx$ и $c + dx$. Директан приступ подразумева израчунавање $ac + (ad + bc)x + bd$, што подразумева 4 множења. Карацубина кључна опаска је да се исто може остварити само са три множења (на рачун мало већег броја сабирања тј. одузимања, што није критично, јер са сабирање и одузимање обично врши брже него множење, а што важи и за полиноме, јер је сабирање и одузимање полинома операција линеарне сложености). Наиме, важи да је $ad + bc = (a + b)(c + d) - (ac + bd)$. Потребно је, дакле, само израчунати производе ac, bd и $(a + b)(c + d)$, а онда прва два производа употребити по два пута (они су потребни и директно и за израчунавање производа $ad + bc$).

Имајући овај Карацубин "трик" у виду, лако можемо направити алгоритам заснован на декомпозицији. Да би имплементација била једноставнија пре множења полиноме допуњујемо нулама тако да оба имају 2^n коефицијената.

На пример, два полинома степена 3 (са по 4 коефицијената) множимо тако што их представимо помоћу полинома 1 (са по 2 коефицијента).

$$(a_3x^3 + a_2x^2 + a_1x + a_0) \cdot (b_3x^3 + b_2x^2 + b_1x + a_0) = ((a_3x + a_2)x^2 + (a_1x + a_0)) \cdot ((b_3x + b_2)x^2 + (b_1x + b_0))$$

Сада су "коефицијенти" $A = a_3x + a_2$ и $B = a_1x + a_0$, $C = b_3x + b_2$ и $D = b_1x + b_0$ и врши се множење полинома $Ax^2 + B$ и $Cx^2 + D$. На основу Карацубиног поступка рекурзивно ћемо израчунавати производе $(A + B)(C + D)$, AC и BD, а то су сада потпроблеми двоструко мање димензије од полазне. Аналогно се поступа и када је димензија било који већи степен двојке.

Пошто множење задовољава једначину $T(n) = 3T(n/2) + O(n)$. Заиста, врше се три множења двоструко мањих полинома, док се сва потребна сабирања (и у фази припреме рекурзивних позива и у фази обједињавања

њихових резултата) врше у сложености $O(n)$. Сложеност алгоритма је зато $O(n^{\log_2 3})$.

Имплементацију најједноставније можемо направити тако да се у сваком рекурзивном позиву сви међурезултати, као и крајњи резултат смештају у посебним векторима. Међутим, таква имплементација је неефикасна и тестови показују да не доприноси побољшању ефикасности наивне процедуре. Кључни проблем је то што се током рекурзије граде вектори у којима се чувају привремени резултати и те алокације и деалокације троше јако пуно времена.

```
// funkcija mnozi dva polinoma p1*p2 sa po 2^k koeficijenata
vector<double> karacuba(const vector<double>& p1,
                        const vector<double>& p2) {
  // broj koeficijenata polinoma
  int n = p1.size();

  // polinome stepena 0 direktno množimo
  if (n == 1)
    return vector<double>{p1[0] * p2[0], 0.0};

  // delimo p1 na dve polovine: a i b
  vector<double> a(n / 2), b(n / 2);
  copy_n(begin(p1), n/2, begin(a));
  copy_n(next(begin(p1), n/2), n/2, begin(b));

  // delimo p2 na dve polovine: c i d
  vector<double> c(n / 2), d(n / 2);
  copy_n(begin(p2), n/2, begin(c));
  copy_n(next(begin(p2), n/2), n/2, begin(d));

  // Važi:
  //   (ax+b)*(cx+d) = a*c*x^2 + ((a+b)*(c+d) - a*c - b*d)*x + b*d

  // rekurzivno računamo a*c i b*d
  vector<double> ac = karacuba(a, c);
  vector<double> bd = karacuba(b, d);

  // izračunavamo a+b (rezultat smeštamo u vektor a)
  for (int i = 0; i < n/2; i++)
    a[i] += b[i];
  // izračunavamo c+d (rezultat smeštamo u vektor c)
  for (int i = 0; i < n/2; i++)
    c[i] += d[i];

  // izračunavamo (a+b)*(c+d)
  vector<double> adbc = karacuba(a, c);
  // izračunavamo (a+b)*(c+d) - a*c - b*d
  for (int i = 0; i < n; i++)
    adbc[i] -= ac[i] + bd[i];

  // sklapamo proizvod iz delova
  vector<double> proizvod(2*n, 0.0);
  for (int i = 0; i < n; i++) {
    proizvod[n + i] += bd[i];
    proizvod[n/2 + i] += adbc[i];
    proizvod[i] += ac[i];
  }

  // vraćamo rezultat
  return proizvod;
}
```

```
// najmanji broj oblika 2^k koji je veci ili jednak od n
int stepenDvojke(int n) {
  int s = 1;
  while (s < n)
    s <<= 1;
  return s;
}

// funkcija mnozi dva polinoma, prosirujuci ih eventualno nulama
vector<double> mnozi_polinome(vector<double>& p1,
                              vector<double>& p2) {
  int s1 = stepenDvojke(p1.size());
  int s2 = stepenDvojke(p2.size());
  int s = max(s1, s2);
  p1.resize(s, 0.0); p2.resize(s, 0.0);
  return karacuba(p1, p2);
}
```

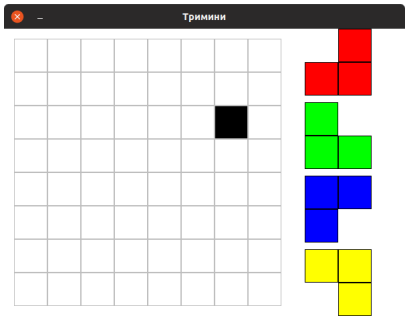
Пажљивија анализа показује да је могуће сву помоћну меморију алоцирати само једном и онда током рекурзије користити стално исти помоћни меморијски простор. Величина потребне помоћне меморије је $4n$ (два пута по n да се сместе полиноми $a+b$ и $c+d$ и још $2n$ да се смести њихов производ). Додатна оптимизација је да се примети да је за мале степене полинома класичан алгоритам бржи него алгоритам заснован на декомпозицији (ово је чест случај код алгоритама заснованих на декомпозицији). Експерименталном анализом се утврђује да се више исплати применити класичан алгоритам кад год је $n \leq 4$.

```
// množimo polinome čiji su koeficijenti smešteni u vektorima
// p1[start1, start1+n) i p2[start2, start2+n)
// i rezultat smeštamo u vektor
// proizvod[start_proizvod, start_proizvod + 2n),
// koristeći pomocni memorijski prostor u vektoru
// pom[start_pom, start_pom + 4n)
void karacuba(int n,
              const vector<double>& p1, int start1,
              const vector<double>& p2, int start2,
              vector<double>& proizvod, int start_proizvod,
              vector<double>& pom, int start_pom) {

  // izlaz iz rekurzije
  if (n <= 4) {
    // klasični algoritam množenja
    for (int i = 0; i < 2*n; i++)
      proizvod[start_proizvod + i] = 0;
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
        proizvod[start_proizvod + i+j] +=
           p1[start1 + i] * p2[start2 + j];
    return;
  }

  // Važi: (a+bx)*(c+dx) =
  //           a*c + ((a+b)*(c+d) - a*c - b*d)*x + b*d*x^2

  // Izračunavamo rekurzivno a*c i smeštamo ga u levu polovinu
  // proizvoda
  karacuba(n / 2, p1, start1, p2, start2,
           proizvod, start_proizvod, pom, start_pom);
  // Izračunavamo rekurzivno b*d i smeštamo ga u desnu polovinu
  // proizvoda
```

```
  karacuba(n / 2, p1, start1 + n/2, p2, start2 + n/2,
           proizvod, start_proizvod + n, pom, start_pom);

  // Izračunavamo a+b i smeštamo ga u pomoćni vektor (na početak)
  for (int i = 0; i < n/2; i++)
    pom[start_pom + i] =
        p1[start1 + i] + p1[start1 + n/2 + i];
  // Izračunavamo c+d i smeštamo ga u pomoćni vektor (iza (a+b))
  for (int i = 0; i < n/2; i++)
    pom[start_pom + n / 2 + i] =
        p2[start2 + i] + p2[start2 + n/2 + i];

  // Rekurzivno izračunavamo (a+b)*(c+d) i smeštamo ga
  // u pomoćni vektor, iza (a+b) i (c+d)
  karacuba(n / 2, pom, start_pom, pom, start_pom + n / 2,
           pom, start_pom + n, pom, start_pom + 2*n);

  // Izračunavamo (a+b)*(c+d) - (ac + bd)
  for (int i = 0; i < n; i++)
    pom[start_pom + n + i] -=
      proizvod[start_proizvod + i] + proizvod[start_proizvod + n + i];

  // Dodajemo ad+bc na sredinu proizvoda
  for (int i = 0; i < n; i++)
    proizvod[start_proizvod + n/2 + i] += pom[start_pom + n + i];
}

// funkcija množi dva polinoma p1*p2 sa po 2^k koeficijenata
vector<double> karacuba(const vector<double>& p1,
                        const vector<double>& p2) {
  int n = p1.size();
  // koeficijenti proizvoda
  vector<double> proizvod(2 * n);
  // pomoćni memorijski prostor potreban za realizaciju algoritma
  vector<double> pom(4 * n);
  // vršimo množenje
  karacuba(n, p1, 0, p2, 0, proizvod, 0, pom, 0);
  // vraćamo proizvod
  return proizvod;
}
```

Задатак: Тримини

Нека је дата табла димензије 8×8 на којој недостаје једно поље. Задатак је попунити преостала 63 поља триминима (облицима који се добију када се из квадрата димензије 2×2 избаци једно поље).



Слика 5.5: Тримини

Улаз: Са стандардног улаза се учитавају два цела броја између 0 и 7, раздвојена размаком која представљају координате (врсту и колону) поља које је избачено са табле.

Излаз: На стандардни излаз исписати матрицу карактера која представља таблу прекривену триминима. На месту недостајућег поља треба да буде исписан размак, а тримини се представљају различитим карактерима (на пример, малим словима енглеске абецеде). Решење није јединствено.

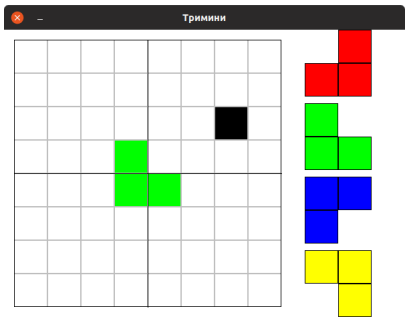
Пример

Улаз	*Излаз*

`3 5`	`cceemmoo`
	`cbbemllo`
	`dbffnnlp`
	`ddfan pp`
	`hhjaartt`
	`hgjjrrqt`
	`iggksqqu`
	`iikkssuu`

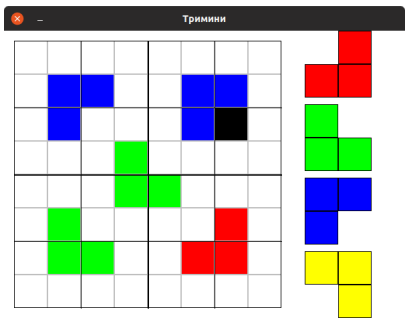
Решење

Задатак се може решити наредним рекурзивним поступком. Табла димензије 8×8 се може разложити на 4 табле димензије 8×8. Постављањем једног тримина можемо постићи да у свакој од те 4 табле недостаје по један тримино.



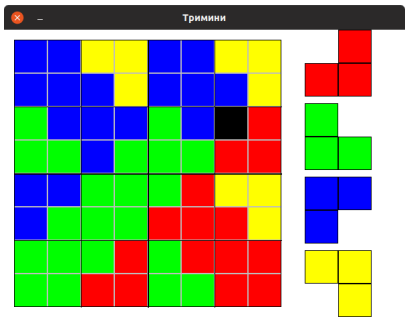
Слика 5.6: Тримини

Након тога, сваки од 4 потпроблема решавамо рекурзивно. Свака табла димензије 4×4 се може разложити на 4 табле димензије 2×2. Постављањем једног тримина можемо постићи да у свакој од те 4 табле недостаје по један тримино.



Слика 5.7: Тримини

На крају, сваку таблу димензије 2×2 којој недостаје једно поље можемо једноставно попунити постављањем одговарајућег тримина.



Слика 5.8: Тримини

Рекурзивној функцији је уз тренутно стање табле и број наредног тримина који се поставља (та променљива се прослеђује по референци, јер се током рекурзије мења и представља улазно-излазни параметар функције) морају проследити аргументи који описују правоугаоник који се тренутно попуњава и положај недостајућег (већ попуњеног) поља. Правоугаоник се описује координатама (бројем врсте и колоне) његовог горњег левог темена и димензијом.

```
// popunjava se kvadrat dimenzije dim cije je gornje levo teme na
// poziciji (v0, k0) i kome je polje na poziciji (v, k) vec popunjeno
// broj je redni broj narednog trimina koji se moze postaviti koji
// se tokom popunjavanja uvecava
void trimini(int tabla[8][8], int dim, int v0, int k0,
             int v, int k, int& broj)
{
  // ako je dimenzija kvadrata 1x1 u kome je jedno polje popunjeno,
  // tada je ceo kvadrat vec popunjen
  if (dim > 1) {
    // kordinate 4 polja oko sredista kvadrata
    int vsred1 = v0 + dim/2 - 1;
    int vsred2 = v0 + dim/2;
    int ksred1 = k0 + dim/2 - 1;
    int ksred2 = k0 + dim/2;

    // odredjujemo kvadrant kome pripada postojeca rupa
    bool rupaGoreLevo = v <= vsred1 && k <= ksred1;
    bool rupaDoleLevo = v >= vsred2 && k <= ksred1;
    bool rupaGoreDesno = v <= vsred1 && k >= ksred2;
    bool rupaDoleDesno = v >= vsred2 && k >= ksred2;

    // postavljamo trimino u sredisna polja kvadrata
    if (!rupaGoreLevo)
      tabla[vsred1][ksred1] = broj;
    if (!rupaDoleLevo)
      tabla[vsred2][ksred1] = broj;
    if (!rupaGoreDesno)
      tabla[vsred1][ksred2] = broj;
    if (!rupaDoleDesno)
      tabla[vsred2][ksred2] = broj;
    broj++;

    // rekurzivno popunjavamo 4 manja kvadrata

    // uvek ispitujemo da li je u tom kvadrantu stara rupa ili
    // je nastala postavljanjem novog trimina

    // kvadrat gore levo - gornje levo teme mu je na (v0, k0)
```

```
    // a nova rupa mu je u njegovom donjem desnom temenu na (vsred1, ksred1)
    if (rupaGoreLevo)
      trimini(tabla, dim/2, v0, k0, v, k, broj);
    else
      trimini(tabla, dim/2, v0, k0, vsred1, ksred1, broj);

    // kvadrat dole levo - gornje levo teme mu je na (vsred2, k0)
    // a nova rupa mu je u njegovom gornjem desnom temenu na (vsred2, ksred1)
    if (rupaDoleLevo)
      trimini(tabla, dim/2, vsred2, k0, v, k, broj);
    else
      trimini(tabla, dim/2, vsred2, k0, vsred2, ksred1, broj);

    // kvadrat gore desno - gornje levo teme mu je na (v0, ksred2)
    // a nova rupa mu je u njegovom donjem levom temenu na (vsred1, ksred2)
    if (rupaGoreDesno)
      trimini(tabla, dim/2, v0, ksred2, v, k, broj);
    else
      trimini(tabla, dim/2, v0, ksred2, vsred1, ksred2, broj);

    // kvadrat dole desno - gornje levo teme mu je na (vsred2, ksred2)
    // i nova rupa mu je u tom gornjem levom temenu na (vsred2, ksred2)
    if (rupaDoleDesno)
      trimini(tabla, dim/2, vsred2, ksred2, v, k, broj);
    else
      trimini(tabla, dim/2, vsred2, ksred2, vsred2, ksred2, broj);
  }
}

// triminima popunjavamo tablu dimenzije dim kome nedostaje polje (v, k)
void trimini(int dim, int v, int k, int tabla[8][8])  {
  // popunjavamo pocetno polje
  int broj = 0;
  tabla[v][k] = broj++;
  // rekurzivnom funkcijom popunjavamo ostatak table
  trimini(tabla, dim, 0, 0, v, k, broj);
}
```