Čas 9.1, 9.2, 9.3 - dekompozicija (divide-and-conquer)

U mnogim situacijama efikasni algoritmi se mogu dobiti time što se niz podeli na dva dela koji se nezavisno obrađuju i nakon toga se konačni rezultat dobija objedinjavanjem tako dobijenih rezultata. Ova tehnika se naziva tehnika *razlaganja*, tehnika *dekompozicije* ili tehnika *podeli-pa-vladaj* (engl. divide-and-conquer). Ako su delovi koji se obrađuju jednaki, dobija se jednačina $T(n) = 2T(n/2) + O(n)$, $T(0) = O(1)$, čije je rešenje $O(n \log n)$, jednačina $T(n) = 2T(n/2) + O(1)$, $T(0) = O(1)$ čije je rešenje $O(n)$ ili jednačina $T(n) = 2T(n/2) + O(\log n)$, $T(0) = O(1)$ čije je rešenje $O(n)$. Treba obratiti pažnju na to da ako su polovine neravnomerne, moguće je da se dobije proces koji se opisuje jednačinom $T(n) = T(n-1) + O(n)$, $T(0) = O(1)$, čije je rešenje $O(n^2)$. Osnovni primere te tehnike su sortiranje objedinjavanjem (engl. Merge Sort) i brzo sortiranje (engl. Quick Sort).

U nekim slučajevima nije neophodno rešavati oba potproblema. Tipičan primer je algoritam binarne pretrage. Jednačina koja se u tom slučaju dobija je $T(n) = T(n/2) + O(1)$, $T(0) = O(1)$ čije je rešenje $O(\log n)$ ili $T(n) = T(n/2) + O(n)$, $T(0) = O(1)$, čije je rešenje $O(n)$.

U nastavku ćemo prikazati nekoliko primera zadataka rešenih ovom tehnikom.

Merge-sort

Problem: Implementirati sortiranje niza objedinjavanjem sortiranih polovina.

```
// 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);
}
```

Broj invezija u nizu

Problem: Odredi koliko različitih parova elemenata u nizu je takvo da je prvi element strogo veći drugog. Na primer, u nizu $5, 4, 3, 1, 2$ takvi su parovi $(5, 4)$, $(5, 3)$, $(5, 1)$, $(5, 2)$, $(4, 3)$, $(4, 1)$, $(4, 2)$, $(3, 1)$ i $(3, 2)$ i ima ih 9.

Jedan način da se odredi broj inverzija je da se niz sortira sortiranjem objedinjavanjem, prilagođenim tako da se broje inverzije. Rekurzivno određujemo broj inverzija u levoj i desnoj polovini niza. Nakon toga, prilikom objedinjavanja određujemo broj inverzija tako da je prvi element u levoj, a drugi u desnoj polovini niza. Primenjujemo klasičan algoritam objedinjavanja (zasnovan na tehnici dva pokazivača) i ako se dogodi da je tekući element u desnoj polovini niza strogo manji od tekućeg elementa u levoj polovini niza, znamo da je strogo

manji i od svih elemenata u levoj polovini iza njega. Znači taj element učestvuje u onoliko inverzija koliko je još ostalo elemenata u levoj polovini niza. Pošto se niz tokom algoritma sortira, ako želimo da zadržimo njegov originalni sadržaj, potrebno ga je pre primene algoritma prekopirati u pomoćni niz.

```
int broj_inverzija(vector<int>& a, int l, int d, vector<int>& b) {
  if (l >= d)
    return 0;
  int s = l + (d - l) / 2;
  int 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++];

  for (int i = l; i <= d; i++)
     a[i] = b[i-l];

  return broj;
}

int broj_inverzija(const vector<int>& a) {
  vector<int> pom1(a.size()), pom2(a.size());
  for (int i = 0; i < a.size(); i++)
     pom1[i] = a[i];
  return broj_inverzija(pom1, 0, pom1.size()-1, pom2);
}
```

Quick-sort

Problem: Sortirati niz brojeva primenom algoritma brzog sortiranja.

```
// 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);
}
```

Quick-select

Problem: U nizu od n elemenata pronaći element od kojega je tačno k elemenata manje ili jednako.

Jedno rešenje je da se ceo niz sortira i da se onda vrati element na poziciji k. Međutim, ovo može biti neefikasno. Zamislimo, da je k mali broj, na primer 0. Tada se traži minimim niza, a rešenje zasnovano na sortiranju nepotrebno određuje međusobni odnos svih elemenata iza njega. Umesto algoritma složenosti $O(n)$ koristimo algoritam složenosti $O(n \log n)$.

Rešenje koje je efikasnije od sortiranja se zasniva na modifikaciji algoritma QuickSort koja je poznata pod imenom *QuickSelect*. Pokažimo varijantu koja određuje element niza na poziciji k (koja se nalazi u intervalu $[l, d]$, gde su l i d granice dela niza koji se obrađuje.

Isto kao i u slučaju algoritma QuickSort, algoritam QuickSelect počinje odabirom pivotirajućeg elementa i particionisanjem niza. Nakon particionisanja postoje tri mogućnosti. Neka je pozicija pivota nakon particionisanja p. Jedna je da se pivot nalazi baš na traženoj poziciji n, tj. da je $p = k$ i u tom slučaju funkcija vraća pivot i završava sa radom. Druga mogućnost je da je $k < p$ i tada se algoritam primenjuje na deo niza $[l, p-1]$. Na kraju, treća mogućnost je da je $k > p$ i tada se algoritam primenjuje na deo niza $[p+1, d]$. Osnovna implementacija može biti rekurzivna (po uzoru na QuickSort funkcija prima parametre l i d),

međutim, pošto se u varijanti QuickSelect vrši samo jedan rekurzivni poziv i to kao repni, rekurziju je veoma jednostavno eliminisati.

Kao i u slučaju algoritma QuickSort i složenost algoritma QuickSelect zavisi od toga koliko sreće imamo da pivot ravnomerno podeli interval $[l, d]$. Ako bi se u svakom koraku desilo da pivot upadne na sredinu intervala, ukupan broj poređenja i razmena bio bi $O(n)$. Zaista, ponašanje funkcije bismo mogli opisati jendačinom $T(n) = T(n/2) + O(n)$, $T(0) = O(1)$, čije je rešenje $O(n)$. U prvom koraku se prilikom pivotiranja vrši n poređenja, u drugom $n/2$, u trećem $n/4$ i tako dalje, što se odozgo može ograničiti sa $2n$). Sa druge strane, ako bi pivot stalno bio blizak nekom od dva kraja intervala $[l, d]$, tada bi složenost algoritma bila $O(n^2)$. Zaista, ponašanje bi se tada moglo opisati pomoću $T(n) = T(n-1) + O(n)$, $T(0) = O(1)$. U prvom koraku bismo imali n poređenja, u drugom $n-1$, u trećem $n-2$ što u zbiru daje $n(n+1)/2$. Sličnom analizom kao onom kojom smo pokazali da je prosečna složenost algoritma QuickSort jednaka $O(n \log n)$, može se pokazati da je prosečna složenost algoritma QuickSelect jednaka $O(n)$.

Pomenimo da izbor algoritma particionisanja može uticati na efikasnost, naročito u slučaju kada u nizu ima dosta ponovljenih elemenata. Algoritam koji particionisanje vrši tako što niz obilazi sa dva kraja i vrši razmene je u tom svetlu dosta efikasniji od algoritma koji niz obilazi samo sa jednog kraja.

```
// pronalazimo
int ktiElement(vector<int>& a, int l, int d, int k) {
  while (true) {
    // pivot dovodimo na poziciju l
    swap(a[l], a[random_value(l, d)]);
    // particionišemo elemente niza
    int i = l + 1, j = d;
    while (i <= j) {
      if (a[i] < a[l])
         i++;
      else if (a[j] > a[l])
         j--;
      else
        swap(a[i++], a[j--]);
    }
    // pivot vraćamo na poziciju j
    swap(a[l], a[j]);
    // pre pivota postoji bar k elemenata pa je dovoljno da
    // pretragu nastavimo samo u delu niza pre pivota
    if (k < j)
      d = j - 1;
    // zaključno sa pivoto
    else if (k > j)
      l = j + 1;
    // pivot je tačno k-ti po redu
    else
      return a[k];
```

```
  }
}

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

Na sličan način možemo odrediti i zbir najvećih k elemenata niza.

```
// 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 <= 0 || l >= d)
    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
    // neki kod k najvecih su 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, k) {
  qsortk(a, 0, a.size() - 1, k);
}

int zbirKNajvecih(vector<int>& a, int k) {
  // odredjujemo prvih k najvecih elemenata niza
  qsortK(a, k);

  // sabiramo prvih k elemenata niza i vraćamo rezultat
  int s = 0;
  for (int i = 0; i < k; i++)
    s += a[i];
  return s;
}
```

U jeziku C++ na raspolaganju imamo funkcije `nth_element` i `partial_sort`

koje vrše delimično sortiranje niza. Funkcija `nth_element` organizuje niz tako da je n-ti element na svom mestu i da su svi elementi ispred njega manji (ne obavezno sortirani), dok `partial_sort` obezbeđuje da je sortirano prvih n elemenata (i da su oni manji od elemenata iza n-tog, koji ne moraju biti sortirani).

```
// niz particionisemo tako da je  n-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;
```

Minimum i maksimum niza

Problem: Pretpostavimo da se u nizu nalaze elementi koje je moguće upoređivati, ali čije je poređenje skupa operacija (na primer dugačke niske, koje se porede leksikografski). Definisati funkciju koja efikasno određuje minimum i maksimum tog niza.

Jasno je da složenost mora da bude $O(n)$, ali pošto je operacija poređenja skupa zanima nas i konstanta koja se javlja uz n tj. želimo da smanjimo broj poređenja.

Jedan način da se zadatak reši je da se minimum i maksimum nađu nezavisno i za to nam je potrebno $2n - 2$ operacija poređenja.

Možemo pokušati induktivno-rekurzivnim pristupom. Pretpostavimo da znamo minimum i maksimum svih elemenata niza osim poslednjeg. Maksimum i minimum ažuriramo tako što poslednji element sa tim minimumom i maksimumom i na taj način problem rešavamo sa 2 poređenja po članu niza, tj. opet sa oko $2n-2$ poređenja (jer u baznom slučaju samo inicijalizujemo minimum i maksimum na početni član niza). Rekurentna jednačina u ovom slučaju je $T(n) = T(n-1)+2$, $T(1) = 0$ i njeno je rešenje $2n - 2$. Recimo i da se pitanje da li je tekući element manji od minimuma može preskočiti ako je tekući element veći od maksimuma (jer element ne može istovremeno da bude i strogo veći od maksimuma i strogo manji od minimuma), ali to ne menja asimptotsku složenost najgoreg slučaja (koja u ovoj situaciji nastupa kod opadajućih nizova).

```
void minimax(const vector<string>& a, int& Min, int& Max) {
  Min = 0; Max = 0;
  for (int i = 1; i < a.size(); i++)
    if (a[i] > a[Max])
      Max = i;
    else if (a[i] < a[Min])
      Min = i;
}
```

Malo efikasnije rešenje se može dobiti dekompozicijom. Pretpostavimo da znamo minimum i maksimum dve polovine niza. Sa tačno dva poređenja možemo

dobiti minimum i maksimum celog niza (minimum je manji od dva minimuma, a maksimum je veći od dva maksimuma).

```
void minimax(const vector<string>& a, int l, int d,
             int& Min, int& Max) {
  if (l == d) {
    Min = l; Max = l;
  } else if (l + 1 == d) {
    if (a[l] < a[d]) {
      Min = l;  Max = d;
    } else {
      Min = d;  Max = l;
    }
  } else {
    int s = l + (d-l) / 2;
    int MinL, MaxL, minD, maxD;
    minimax(a, l, s, MinL, MaxL);
    minimax(a, s+1, d, MinD, MaxD);
    Min = a[MinL] < a[MinD] ? minL : minD;
    Max = a[MaxL] < a[MaxD] ? maxD : maxL;
  }
}


void minimax(const vector<string>& a, int& Min, int& Max) {
   minimax(a, 0, a.size() - 1, Min, Max);
}
```

Pretpostavimo da je n stepen broja 2, tj. da postoji k tako da je $n = 2^k$. Ovo rešenje zadovoljava rekurentnu jednačinu $T(n) = 2T(n/2) + O(1)$, $T(1) = 0$, $T(2) = 1$ čije je rešenje, kao što znamo, $O(n)$, međutim, ovde nas zanima malo preciznija analiza tj. konstanta koja se javlja uz n. Važi:

$$T(2^k) = 2T(2^{k-1}) + 2$$

$$\frac{T(2^k)}{2^k} = \frac{T(2^{k-1})}{2^{k-1}} + \frac{1}{2^{k-1}}$$

Smenom $S(k) = \frac{T(2^k)}{2^k}$ dobijamo da je $S(k) = S(k-1) + \frac{1}{2^{k-1}}$ i sumiranjem dobijamo da je $S(k) = \frac{3}{2} - 2^{1-k}$, pa je $T(n) = \frac{3}{2}n - 2$.

Ovo rešenje je stoga efikasnije od prethodnih (mada je asimptotska složenost svih rešenja $O(n)$). Empirijska analiza testiranjem na test primerima pokazuje da nema značajne razlike u vremenu izvršavanja

Maksimalni zbir segmenta

Problem: Definisati efikasnu funkciju koja pronalazi najveći mogući zbir segmenta (podniza uzastopnih elemenata) datog niza brojeva. Proceniti joj složenost.

Na kraju pokažimo još jedan način da rešimo ovaj problem, zasnovan na tehnici razlaganja.

Dekompozicija nam sugeriše da je poželjno da niz podelimo na dva podniza jednake dužine čija rešenja možemo da konstruišemo na osnovu induktivne hipoteze (najčešće rekurzivnim pozivima). Bazu i ovaj put čini slučaj praznog niza, koji sadrži samo prazan segment čiji je zbir nula. Fiksirajmo središnji element niza. Sve segmente niza možemo da grupišemo u tri grupe: segmente koji su u potpunosti levo od središnjeg elementa, segmente koji su u potpunosti desno od središnjeg elementa i segmente koji sadrže središnji element. Najveće zbirove segmenata u prvoj i u drugoj grupi znamo na osnovu induktivne hipoteze. Najveći zbir segmenta u trećoj grupi možemo lako odrediti analizom svih segmenata: krećemo od jednočlanog segmenta koji sadrži samo središnji element i inkrementalno se širimo nalevo dodajući jedan po jedan element i računajući tekući maksimum, a zatim krećemo od maksimalnog segmenta proširenog nalevo i inkrementalno ga proširujemo jednim po jednim elementom nadesno, tražeći novi maksimum.

```
int maksZbirSegmenta(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(int a[], int n) {
  return maksZbirSegmenta(a, 0, n - 1);
}
```

Analiza složenosti ovaj put zahteva kompleksniji matematički aparat, ali je prilično jednostavna kada se taj aparat poznaje. Naime, ako sa n označimo dužinu niza $d - l + 1$ i ako vreme izvršavanja obeležimo sa $T(n)$, tada važi da je $T(0) = O(1)$ i da je $T(n) = 2T(n/2) + O(n)$. Naime, vrše se dva rekurzivna poziva za duplo manje nizove, a najveći zbir segmenata koji obuhvataju središnji element izračunavamo u vremenu $O(n)$ (što je prilično očigledno jer imamo dve petlje koje se ukupno izvršavaju n puta, a čija su tela konstantne složenosti). Na osnovu master teoreme lako se zaključuje da je $T(n) = O(n \log n)$. Dakle, ovaj algoritam je manje efikasan od prethodna dva, ali je i dalje prilično upotrebljiv, jer je mnogo bolji od početna dva veoma naivna pokušaja.

Postoje i drugi algoritmi složenosti $O(n)$ za rešavanje ovog problema (na primer, onaj dobijen odsecanjima u pretrazi, Kadanov algoritam ili agoritam dobijen preko prefiksnih zboriva), međutim na nizovima manjim od milion elemenata se ne može primetiti razlika između njih i između prethodnog algoritma složenosti $O(n \log n)$. Fine razlike između ovih algoritama se vide tek na nizu od 10 miliona elemenata i tada sve tri implementacije složenosti $O(n)$ posao završavaju za oko $0,120$ sekundi, a ona zasnovana na dekompoziciji za oko $0,330$ sekundi.

Ipak, na ideji dekompozicije možemo izgraditi i efikasniji algoritam. Ključni uvid je da se najveći zbir segmenta oko srednjeg elementa može dobiti kao zbir najvećeg sufiksa niza levo od tog elementa i najvećeg prefiksa niza desno od tog elementa. Možemo ojačati induktivnu hipotezu i umesto da prefiks i sufiks računamo u petlji, u linearnom vremenu, možemo pretpostaviti da za obe polovine niza prefiks i sufiks dobijamo kao rezultat rekurzivnog poziva. To nam je dovoljno da odredimo maksimalni zbir funkcije, ali moramo vratiti dug i naša funkcija sada pored maksimalnog zbira segmenta mora izračunati i maksimalni zbir prefiksa i maksimalni zbir sufiksa celog niza. Maksimalni zbir prefiksa celog niza je veći broj od maksimalnog zbira prefiksa levog dela i od zbira celog levog dela i maksimalnog zbira prefiksa desnog dela. Slično, maksimalni zbir sufiksa celog niza je veći od maksimalnog zbira sufiksa desnog dela i od zbira maksimalnog zbira sufiksa levog dela i celog desnog dela. Zato je neophodno dodatno ojačati induktivnu hipotezu i tokom rekurzije računati i zbir celog niza.

```
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;
}
```

Složenost prethodne implementacije zadovoljava jednačinu $T(n) = 2T(n/2)+O(1)$ i jednaka je $O(n)$.

Silueta zgrada - skyline

Problem: Sa broda se vide zgrade na obali velegrada. Duž obale je postavljena koordinatna osa i za svaku zgradu se zna pozicija levog kraja, visina i pozicija desnog kraja. Napisati program koji izračunava siluetu grada.

Svaku zgradu predstavljaćemo strukturom koja sadrži levi kraj zgrade a, zatim desni kraj zgrade b i njenu visinu 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 deo-po-deo konstantna funkcija i određena je intervalima konstantnosti $(-\infty, x_0)$, $[x_0, x_1)$, $[x_1, x_2)$, $\ldots$, $[x_{n-1}, +\infty)$, određenim tačkama podele $x_0 < x_1 < \ldots < x_{n-1}$ i vrednostima 0, h_0, $\ldots$, h_{n-2} i 0$ funkcije na svakom od intervala.

Svaki uređeni par (x_i, h_i) predstavljaćemo strukturom promena, a siluetu ćemo predstavljati vektorom promena.

```
struct promena {
  int x, h;
  promena(int x = 0, int h = 0)
    : x(x), h(h) {
  }
};
```

```
vector<promena> silueta;
```

Direktan induktivno-rekurzivni pristup bi podrazumevao da se prvo napravi silueta svih zgrada osim poslednje i da se onda ažurira na osnovu poslednje zgrade (bazni slučaj je silueta jedne zgrade, koja se trivijalno određuje). Vreme potrebno da se zgrada umetne u siluetu zavisi od dužine siluete koja je u najgorem slučaju jednaka broju obrađenih zgrada. Ovim se dobija jednačina $T(n) = T(n-1) + O(n)$, $T(1) = O(1)$, čije je rešenje $O(n^2)$.

Problem možemo efikasnije rešiti tehnikom podeli-pa-vladaj. Ključna opaska je to da dve siluete možemo objediniti za isto vreme za koje možemo objediniti jednu zgradu u siluetu. Pošto su siluete sortirane možemo ih obilaziti uz održavanje dva pokazivača i objedinjavati veoma slično objedinjavanju dva sortirana niza brojeva.

```
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 {
        x = xd; Hd = rezultat_d[dd].h;
        dd++;
      }
    }

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

    // integrišemo (x, h) u tekuću rezultujući siluetu
    if (rezultat.size() > 0) {
      // poslednja promena tekuće siluete
      int xb = rezultat.back().x, hb = rezultat.back().h;
      // ako se promena dešava na istoj x koordinati kao prethodna
      // samo menjamo visinu
      if (x == xb)
        rezultat.back().h = h;
      // ako nova promena ima istu visinu kao prethodna, preskačemo je
      else if (h != hb)
        // u suprotnom dodajemo novu promenu u niz
        rezultat.emplace_back(x, h);
    } else
      // nema prethodne promene, pa novu promenu dodajemo na početak niza
      rezultat.emplace_back(x, h);
  }

  return rezultat;
}

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

Karacubin algoritam množenja polinoma

Problem: Definisati funkciju koja množi dva polinoma predstavljena vektorima svojih koeficijenata. Jednostavnosti radi pretpostaviti da su vektori dužine 2^k.

Klasičan algoritam množenja koji se uči u srednjoj školi ima složenost $O(n^2)$.

```
// funkcija mnozi dva polinoma p1*p2
vector<double> proizvod(const vector<double>& p1,
                        const vector<double>& p2) {
  vector<double> proizvod(2*n-1, 0);
```

```
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
      proizvod[i+j] += p1[i] * p2[j];
  return c;
}
```

Iako se neko vreme smatralo da je to donja granica složenosti, Anatolij Karacuba je 1960. pokazao da je dekompozicijom moguće dobiti efikasniji algoritam. Pretpostavimo da je potrebno pomnožiti polinome $a + bx$ i $c + dx$. Direktan pristup podrazumeva izračunavanje $ac + (ad + bc)x + bd$, što podrazumeva 4 množenja. Karacubina ključna opaska je da se isto može ostvariti samo sa tri množenja (na račun malo većeg broja sabiranja tj. oduzimanja, što nije kritično, jer sa sabiranje i oduzimanje obično vrši brže nego množenje, a što važi i za polinome, jer je sabiranje i oduzimanje polinoma operacija linearne složenosti). Naime, važi da je $ad + bc = (a + b)(c + d) - (ac + bd)$. Potrebno je, dakle, samo izračunati proizvode ac, bd i $(a + b)(c + d)$, a onda prva dva proizvoda upotrebiti po dva puta (oni su potrebni i direktno i za izračunavanje proizvoda $ad + bc$).

Imajući ovaj Karacubin "trik" u vidu, lako možemo napraviti algoritam zasnovan na dekompoziciji.

```
// 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(1, p1[0] * p2[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 (koristimo pomoćni vektor a)
```

```
  for (int i = 0; i < n/2; i++)
    a[i] += b[i];
  // izračunavamo c+d (koristimo pomoćni vektor b)
  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;
}
```

Složenost prethodnog algoritma je određena rekurentnom jednačinom $T(n) = 3T(n/2) + O(n)$ i iznosi $O(n^{\log_3 2})$.

Međutim, prethodna implementacija je neefikasna i testovi pokazuju da ne doprinosi poboljšanju efikasnosti naivne procedure. Ključni problem je to što se tokom rekurzije grade vektori u kojima se čuvaju privremeni rezultati i te alokacije i dealokacije troše jako puno vremena. Pažljivija analiza pokazuje da je moguće svu pomoćnu memoriju alocirati samo jednom i onda tokom rekurzije koristiti stalno isti pomoćni memorijski prostor. Veličina potrebne pomoćne memorije je $4n$ (dva puta po n da se smeste polinomi $a + b$ i $c + d$ i još $2n$ da se smesti njihov proizvod). Dodatna optimizacija je da se primeti da je za male stepene polinoma klasičan algoritam brži nego algoritam zasnovan na dekompoziciji (ovo je čest slučaj kod algoritama zasnovanih na dekompoziciji). Eksperimentalnom analizom se utvrđuje da se više isplati primeniti klasičan algoritam kad god je $n \leq 4$.

```
// 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;
}
```

FFT - brza Furijeova transformacija

Još brži način množenja dva polinoma zasnovan je na čuvenoj *brzoj Furijeovoj transformaciji*, koja je korisna i u mnogim drugim kontekstima. Lep opis ove transformacije dostupan je u knjizi profesora Miodraga Živkovića.

Razmotrimo ovde pitanje implementacije ovog algoritma. U jeziku C++ kompleksne brojeve imamo na raspolaganju u obliku tipova `complex<double>` i `complex<float>` (zapis u dvotrukoj i jednostrukoj tačnosti). Direktan način implementacije je sledeći.

```
typedef complex<double> Complex;
typedef vector<Complex> ComplexVector;

// brza Furijeova transformacija vektora a duzine n=2^k
// bool parametar inverzna odredjuje da li je direktna ili inverzna
ComplexVector fft(const ComplexVector& a, bool inverzna) {
  // broj koeficijenata polinoma
  int n = a.size();

  // ako je stepen polinoma 0, vrednost u svakoj tacki jednaka
  // je jedinom koeficijentu
  if (n == 1)
    return ComplexVector(1, a[0]);

  // izdvajamo koeficijente na parnim i na neparnim pozicijama
  ComplexVector A(n / 2), B(n / 2);
  for (int i = 0; i < n / 2; i++) {
    A[i] = a[2 * i];
    B[i] = a[2 * i + 1];
  }

  // rekurzivno izracunavamo Furijeove transformacije tih polinoma
```

```
  ComplexVector fftA = fft(A, inverzna),
                fftB = fft(B, inverzna);

  // objedinjujemo rezultat
  ComplexVector rez(n);
  for (int k = 0; k < n; k++) {
    // odredjujemo primitivni n-ti koren iz jedinice
    double coeff = inverzna ? -1.0 : 1.0;
    complex<double> w = exp((coeff * 2 * k * M_PI / n) * 1i);
    // racunamo vrednost polinoma u toj tacki
    rez[k] = fftA[k % (n / 2)] + w * fftB[k % (n / 2)];
  }
  // vracamo konacan rezultat
  return rez;
}


// funkcija vrsi direktnu Furijeovu transformaciju polinoma ciji su
// koeficijenti odredjeni nizom a duzine 2^k
ComplexVector fft(const ComplexVector& a) {
  return fft(a, false);
}

// funkcija vrsi inverznu Furijeovu transformaciju polinoma ciji su
// koeficijenti odredjeni nizom a duzine 2^k
ComplexVector ifft(const ComplexVector& a) {
  ComplexVector rez = fft(a, true);
  // nakon izracunavanja vrednosti, potrebno je jos podeliti
  // sve koeficijente duzinom vektora
  int n = a.size();
  for (int k = 0; k < n; k++)
    rez[k] /= n;
  return rez;
}
```

Jasno je da prethodna procedura zadovoljava jednačinu $T(n) = 2T(n/2) + O(n)$, pa joj je složenost $O(n \log n)$. Nakon transformacije (promene reprezentacije izračunavanjem vrednosti) dva polinoma njihovo množenje je moguće u vremenu $O(n)$, pa je ukupna složenost množenja polinoma pomoću FFT jednaka $O(n \log n)$.

Međutim, prethodnu naivnu implementaciju je moguće poboljšati. Najveći problem je to što se primitivni n-ti koreni iz jedinice računaju zasebno u svakom rekurzivnom pozivu. Efikasnost bi se značajno popravila ako bi se niz korena izračunao samo jednom, pre funkcije. Problem je u tome što se tokom rekurzije n smanjuje, pa su nam u svakom pozivu potrebni različiti koreni. Međutim, ako je neki broj k-ti primitivni koren iz jedinice, onda je on i $2k$-ti koren primitivni koren iz jedinice. Zato, ako znamo niz primitivnih n-tih korena iz jedinice ($e^{\frac{2k\pi i}{n}}$, za k od 0 do $n-1$), tada su elementi na parnim pozicijama tog vektora $n/2$-ti

primitivni koreni iz jedinice. Zaista, ako je $k = 2k'$, tada je $e^{\frac{2k\pi i}{n}} = e^{\frac{2k'\pi i}{\frac{n}{2}}}$ i važi da ako je $0 \leq k < n$, tada je $0 \leq k' < n/2$. Dakle, u početku možemo izračunati niz svih primitivnih n-tih korena iz jedinice i njih koristiti na početnom nivou rekurzije, na narednom nivou rekurzije ćemo koristiti svaki drugi element tog vektora, na narednom svaki četvrti, zatim svaki osmi i tako dalje. Na primer, ako je $n = 8$, početni niz korena je 1, $\frac{\sqrt{2}}{2} + i\frac{\sqrt{2}}{2}$, i, $-\frac{\sqrt{2}}{2} + i\frac{\sqrt{2}}{2}$, -1, $-\frac{\sqrt{2}}{2} - i\frac{\sqrt{2}}{2}$, $-i$, $\frac{\sqrt{2}}{2} - i\frac{\sqrt{2}}{2}$, dok je na narednom nivou rekurzije $n = 4$ i koriste se koreni 1, i, -1, $-i$, dok je na narednom nivou rekurzije $i = 2$ i koriste se 1 i -1.

Još jedan problem prethodne implementacije je to što se tokom rekurzije alociraju i popunjavaju pomoćni vektori, što dovodi do gubitka i vremena i memorije. Furijeovu transformaciju je moguće realizovati i bez korišćenja pomoćne memorije. Na početnom nivou rekurzije koeficijenti ulaznog polinoma su svi dati počenim vektorima koeficijenata. Na narednom se posmatraju elementi na pozicijama 0, 2, 4, ..., $n-2$ i na pozicijama 1, 3, ..., $n-1$. Na narednom se posmatraju elementi na pozicijama 0, 4, 8, $n-4$, zatim elementi na pozicijama 1, 5, ..., $n-3$, zatim elemeni na pozicijama 2, 6, ..., $n-2$, i na kraju elementi na pozicijama 3, 7, ..., $n-1$. Slično se nastavlja i na daljim nivoima rekurzije. Dakle, umesto formiranja pomoćnog ulaznog vektora sa pogodno odabranim ulaznim koeficijentima, prosleđivaćemo originalni vektor, poziciju početka s i pomeraj d i posmatraćemo njegove elemente na pozicijama $s + dk$, za $0 \leq k < n$, gde je $n = n_0/d$, a n_0 je dužina početnog vektora. Rezultate rekurzivnih poziva možemo smestiti u dve polovine rezultujućeg niza. Nakon toga rezultate objedinjavamo. Vrednosti na pozicijama k i $k+n/2$ rezultujućeg vektora određene su vrednostima na poziciji k u rezultatu prvog i drugog rekurzivnog, međutim, one se nalaze upravo na pozicijama k i $k + n/2$ rezultujućeg vektora (jer smo rezultate rekurzivnih poziva smestili u prvu i drugu polovinu rezultata). Moramo voditi računa da te dve vrednosti moramo istovremeno izračunati i ažurirati.

```
typedef complex<double> Complex;
typedef vector<Complex> ComplexVector;

// funkcija vrši Furijeovu transformaciju (direkntu ili inverznu) elemenata
//   a[start_a], a[start_a + step], a[start_a + 2step], ...
// i rezultat smešta u niz
//   rez[start_rez], rez[start_rez + 1], rez[start_rez + 2], ...
// koristeci primitivne korene iz jedinice smestene u niz
//   w[0], w[step], w[2step], ...
void fft(const ComplexVector& a, int start_a,
         const ComplexVector& w,
         ComplexVector& rez, int start_rez,
         int step) {
  // broj elemenata niza koji se transformise
  int n = a.size() / step;

  // stepen polinoma je nula, pa mu je vrednost u svakoj tacki jednaka
  // konstantnom koeficijentu
```

```
  if (n == 1) {
    rez[start_rez] = a[start_a];
    return;
  }

  // rekurzivno transformisemo niz koeficijenata na parnim pozicijama
  // smestajuci rezultat u prvu polovinu niza rez
  fft(a, start_a, w, rez, start_rez, step*2);
  // rekurzivno transformisemo niz koeficijenata na neparnim pozicijama
  // smestajuci rezultat u drugu polovinu niza rez
  fft(a, start_a + step, w, rez, start_rez + n/2, step*2);

  // objedinjujemo dve polovine u rezultujuci niz
  for (int i = 0; i < n/2; i++) {
    auto r1 = rez[start_rez + i];
    auto r2 = rez[start_rez + (i + n/2)];
    rez[start_rez + i] = r1 + w[i*step] * r2;
    rez[start_rez + (i + n/2)] = r1 - w[i*step] * r2;
  }
}

// funkcija vrsi direktnu Furijeovu transformaciju polinoma ciji su
// koeficijenti odredjeni nizom a duzine 2^k
ComplexVector fft(const ComplexVector& a) {
  // duzina niza koeficijenata polinoma
  int n = a.size();
  // izracunavamo primitivne n-te korene iz jedinice
  ComplexVector w(n/2);
  for (int k = 0; k < n/2; k++)
    w[k] = exp((2 * k * M_PI / n) * 1i);
  // vektor u koji se smesta rezultat
  ComplexVector rez(n);
  // vrsimo transformaciju
  fft(a, 0, w, rez, 0, 1);
  // vracamo dobijeni rezultat
  return rez;
}

// funkcija vrsi inverznu Furijeovu transformaciju polinoma ciji su
// koeficijenti odredjeni nizom a duzine 2^k
ComplexVector ifft(const ComplexVector& a) {
  // duzina niza koeficijenata polinoma
  int n = a.size();
  // izracunavamo primitivne n-te korene iz jedinice
  ComplexVector w(n);
  for (int k = 0; k < n; k++)
    w[k] = exp((- 2 * k * M_PI / n) * 1i);
  // vektor u koji se smesta rezultat
  ComplexVector rez(n);
  // vrsimo transformaciju
```

```
  fft(a, 0, w, rez, 0, 1);
  // popravljamo rezultat
  for (int i = 0; i < n; i++)
    rez[i] /= n;
  // vracamo dobijeni rezultat
  return rez;
}

vector<double> proizvod(const vector<double>& p1,
                        const vector<double>& p2) {
  // duzina niza koeficijenata
  int n = p1.size();
  // kreiramo nizove kompleksnih koeficijenata dopunjavajuci ih nulama
  // do dvostruke duzine
  int N = 2*n;
  ComplexVector a(N, 0.0);
  copy(begin(p1), end(p1), begin(a));
  ComplexVector b(N, 0.0);
  copy(begin(p2), end(p2), begin(b));

  // vrsimo Furijeove transformacije oba vektora koeficijenata
  ComplexVector va = fft(a), vb = fft(b);
  // prolazimo Furijeovu transformaciju vektora koeficijenata proizvoda
  ComplexVector vc(N);
  for (int i = 0; i < N; i++)
    vc[i] = va[i] * vb[i];
  // inverznom Furijeovom transformacijom rekonstruisemo koeficijente
  // proizvoda
  ComplexVector c = ifft(vc);

  // realne delove kompleksnih brojeva smestamo u niz rez i vracamo ga
  vector<double> rez(N);
  transform(begin(c), end(c), begin(rez),
            [](Complex x){
              return real(x);
            });
  return rez;
}
```

Pažljivom analizom je moguće ukloniti rekurziju iz prethodne implementacije (tako se dobija Kuli-Tukijev nerekurzivni algoritam za FFT, zasnovan na tzv. leptir-shemi), no time se nećemo baviti u sklopu ovog kursa.

Štrasenov algoritam množenja matrica

Problem: Napisati program koji određuje proizvod dve date matrice.

Naivni algoritam za množenje matrica je složenosti $O(n^3)$.

```
typedef vector<vector<int>> Matrica;

void alociraj(Matrica& m, int v, int k) {
  m.resize(v);
  for (int i = 0; i < v; i++)
     m[i].resize(k, 0);
}


Matrica proizvod(Matrica a, Matrica b) {
  Matrica c;
  c.alociraj(v, a.size(), b[0].size());
  for (int i = 0; i < n; i++)
     for (int j = 0; j < n; j++)
        for (int k = 0; k < n; k++)
           c[i][j] += a[i][k] * b[k][j];
  return c;
}
```

Postavlja se pitanje da li je efikasniji algoritam moguće izvesti dekompozicijjom. Ako pretpostavimo da je dimenzija matrice stepen broja 2, tada se oba činioca mogu predstaviti putem četiri podmatrice čije su dimenzije $n/2$. Potrebno je, dakle, izračunati proizvod

$$\left(\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right) = \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) \cdot \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right)$$

n Direktan pristup zahteva izračunavanje proizvoda

$$\begin{aligned} C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\ C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\ C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{aligned}$$

Pošto sabiranje matrica zahteva $O(n^2)$ operacija, ovo rešenje zadovoljava rekurentnu jednačinu $T(n) = 8T(n/2) + O(n^2)$, čije je rešenje, znamo na osnovu master teoreme, jednako $O(n^{\log_2 8}) = O(n^3)$.

Slično kao kod Karacube, ali dosta komplikovanije, Štrasen je primetio da je moguće upotrebiti samo 7 množenja manjih blok matrica. Koristimo 7 pomoćnih matrica.

$$
\begin{aligned}
M_1 &= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}) \\
M_2 &= (A_{2,1} + A_{2,2})B_{1,1} \\
M_3 &= A_{1,1}(B_{1,2} - B_{2,2}) \\
M_4 &= A_{2,2}(B_{2,1} - B_{1,1}) \\
M_5 &= (A_{1,1} + A_{1,2})B_{2,2} \\
M_6 &= (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2}) \\
M_7 &= (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2})
\end{aligned}
$$

I pomoću njih dobijamo 4 tražene matrice.

$$
\begin{aligned}
C_{1,1} &= M_1 + M_4 - M_5 + M_7 \\
C_{1,2} &= M_3 + M_5 \\
C_{2,1} &= M_2 + M_4 \\
C_{2,2} &= M_1 - M_2 + M_3 + M_6
\end{aligned}
$$

Naravno, prethodne formule nije neophodno učiti napamet.

Time dobijamo algoritam koji zadovoljava jednačinu $T(n) = 7T(n/2) + O(n^2)$, čije je rešenje, znamo na osnovu master teoreme jednako $O(n^{\log_2 7}) \approx O(n^{2,8074})$.

Ovaj dobitak na vremenu ima svoju cenu i Štrasenov algoritam ima nekoliko ozbiljnih nedostataka.

- Štrasenov algoritam je komplikovaniji i teži za implementaciju od običnog.
- Kako je to obično slučaj, dekompozicija postaje korisna tek za probleme većih dimenzija. Praktično se pokazuje da je potrebno da dimenzija bude veća od 100, da bi se osetio dobitak. Ova činjenica se koristi tako što se kao baza indukcije tj. izlaz iz rekurzije upotrebi slučaj kada je n manje od oko stotine i tada se primeni naivni algoritam.
- Troši se dodatna memorija, što za početno proširenje matrica tako da im dimenzija bude stepen dvojke, što na smeštanje pomoćnih matrica.
- Štrasenov algoritam je manje numerički stabilan od običnog. Za iste veličine greške ulaznih podataka, Štrasenov algoritam obično dovodi do većih grešaka u izlaznim podacima.
- Štrasenov algoritam se teže paralelizuje.

Najbliži par tačaka

Problem: U ravni je zadato n tačaka svojim koordinatama. Napiši program koji određuje najmanje (Euklidsko) rastojanje među njima.

Rešenje grubom silom podrazumeva ispitivanje svih parova tačaka i složenost mu je $O(n^2)$.

```
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 najbliziParTacaka(const vector<Tacka>& t) {
  double d = numeric_limits<double>::max();
  for (int i = 0; i < n; i++)
    for (int j = i+1; j < n; j++) {
      double dij = rastojanje(tacke[i], tacke[j]);
      if (dij < d)
        d = dij;
    }
  return d;
}
```

Jedan način da se do rešenja dođe efikasnije je da se primeni dekompozicija.

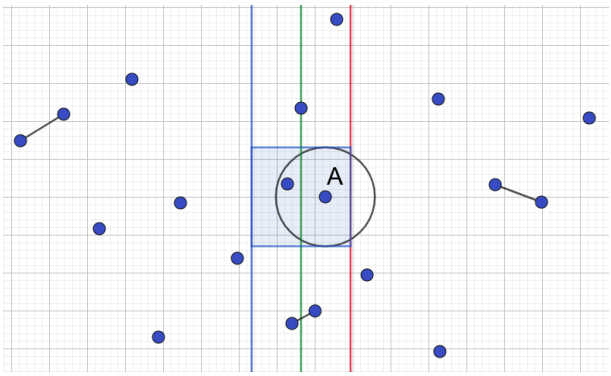
Bazni slučaj predstavlja situacija u kojoj imamo manje od četiri tačke, jer njih ne možemo podeliti u dve polovine u kojima postoji bar po jedan par tačaka (a ako u skupu nemamo bar 2 tačke, najmanje rastojanje nije jasno definisano). U tom slučaju rešenje nalazimo poređenjem rastojanja svih parova tačaka (pošto je tačaka malo, ovaj korak je složenosti $O(1)$).

Skup tačaka možemo jednom vertikalnom linijom podeliti na dve otprilike istobrojne polovine. Ako tačke sortiramo po koordinati x, vertikalna linija može odgovarati koordinati središnje tačke. Rekurzivno određujemo najmanje rastojanje u prvoj polovini (to su tačke levo od vertikalne linije) i u drugoj polovini (to su tačke desno od vertikalne linije). Najbliži par je takav da su (1) obe tačke u levoj polovini, (2) obe tačke u desnoj polovini ili (3) jedna tačka je u levoj, a druga u desnoj polovini. Za prva dva slučaja već znamo rešenja i ostaje da se razmotri samo treći.

Neka je d_l minimalno rastojanje tačaka u levoj polovini, d_r minimalno rastojanje tačaka u desnoj polovini, a d manje od ta dva rastojanja. Ako vertikalna linija ima x-koordinatu x, tada je moguće odbaciti sve tačke koje su levo od $x - d$ i desno od $x + d$, jer je njihovo rastojanje do najbliže tačke iz suprotne polovine sigurno veće od d. Potrebno je ispitati sve preostale tačke, tj. sve tačke iz pojasa $[x - d, x + d]$, proveriti da li među njima postoji neki par tačaka čije je rastojanje strogo manje od d i vrednost d ažurirati na vrednost najmanjeg rastojanja takvog para tačaka. Problem je to što u najgorem slučaju njih može biti puno (moguće

je da se svih n tačaka nađe u tom pojasu) i ako ispitujemo sve parove, dolazimo u najgorem slučaju do oko $n^2/4$ poređenja (ako je pola tačaka levo, a pola desno od linije podele). Ipak, proveru je moguće organizovati tako da se proveri samo mali broj parova tačaka.

Jednostavnosti radi ćemo pretpostaviti da istovremeno razmatramo sve tačke unutar pojasa $[x - d, x + d]$, bez obzira sa koje strane vertikalne linije se nalaze (unapred znamo da je provera tačaka koje su sa iste strane vertikalne linije podele nepotrebna, ali ne može narušiti korektnost, dok god smo sigurni da se porede i svi potrebni parovi tačaka sa različite strane te linije). Svaku tačku A iz pojasa je dovoljno uporediti sa onim tačkama koje leže unutar kruga sa centrom u tački A i poluprečnikom d, što omogućava značajna odsecanja. Pripadnost krugu nije jednostavno proveriti i zato umesto njega možemo razmatrati kvadrat stranice dužine $2d$ na čijoj se horizontalnoj srednjoj liniji nalazi tačka A. Time će odsecanje biti za nijansu manje nego u slučaju kruga, ali će detektovanje tačaka koje pripadaju tom pravougaoniku biti veoma jednostavno. To će biti sve one tačke iz pojasa kojima je koordinata y u intervalu $[y_A - d, y_A + d]$.



Slika 1: Najbliži par tačaka u levom pojasu, desnom pojasu i između pojaseva. Krug i kvadrat kojim su određeni kandidati oko tačke A.

Dalje smanjenje broja poređenja možemo dobiti ako primetimo da svaki par obrađujemo dva puta (jednom dok obrađujemo tačke u okolini prve, a jednom dok obrađujemo tačke u okolini druge tačke). Možemo jednostavno zaključiti da je dovoljno svaku tačku porediti samo sa onim tačkama koje se nalaze na istoj visini kao ona ili iznad nje. Dakle, svaku tačku je potrebno uporediti samo sa tačkama čije x koordinate leže unutar intervala $[x - d, x + d]$ i čije y koordinate leže unutar intervala $[y_A, y_A + d]$. Prvi uslov možemo obezbediti tako što pre poređenja sve tačke iz pojasa širine d oko vertikalne linije podele izdvojimo u poseban niz (za to nam je potrebno $O(n)$ dodatne memorije i vremena). Drugi uslov efikasnije možemo obezbediti ako sve tačke tog pomoćnog niza sortiramo po koordinati y (za to nam je potrebno vreme $O(n \log n)$ i zatim tačke obrađujemo u neopadajućem redosledu y koordinata. Za svaku tačku A obrađujemo samo tačke koje se nalaze iza nje u sortiranom nizu i obrađujemo jednu po jednu tačku

sve dok ne naiđemo na tačku čija je koordinata y veća ili jednaka $y_A + d$ (ona od tačke A ne može biti na manjem rastojanju od d).

Na osnovu prethodne diskusije možemo napraviti sledeću implementaciju.

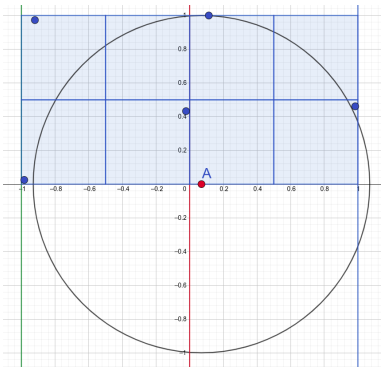
```
bool porediX(const Tacka& t1, const Tacka& t2) {
  return t1.x <= t2.x;
}

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

// funkcija određuje najbliži par tačaka u delu niza [l, r]
double najblizeTacke(vector<Tacka>& tacke, int l, int r) {
  // ako ima manje od 4 tačke, najmanje rastojanje pronalazimo
  // grubom silom
  if (r - l + 1 < 4) {
    // poredimo sve parove tačaka, tražeći minimalno rastojanje
    double d = numeric_limits<double>::max();
    for (int i = l; i < r; i++)
      for (int j = i+1; j <= r; j++) {
        double dij = rastojanje(tacke[i], tacke[j]);
        if (dij < d)
          d = dij;
      }
    return d;
  }

  // pozicija središnje tačke
  int s = l + (r - l) / 2;
  // određujemo najmanje rastojanje tačaka koje su levo od
  // središnje tačke (uključujući i nju)
  double d1 = najblizeTacke(tacke, l, s);
  // određujemo najmanje rastojanje tačaka koje su desno od
  // središnje tačke
  double d2 = najblizeTacke(tacke, s+1, r);
  // određujemo manje od ta dva rastojanja
  double d = min(d1, d2);
  // granice pojasa u kom se nalaze tačke za koje u suprotnom
  // pojasu mogu postajati tačke koje su im bliže od d
  double dl = tacke[s].x - d, dr = tacke[s].x + d;
  // određujemo sve tačke u tom pojasu
  vector<Tacka> pojas;
  for (int i = l; i <= r; i++)
    if (dl <= tacke[i].x && tacke[i].x <= dr)
      pojas.push_back(tacke[i]);
  // sortiramo ih na osnovu koordinate y
  sort(begin(pojas), end(pojas), porediY);
  // obrađujemo sve tačke iz pojasa u rastućem redosledu
```

```
  // koordinata y
  for (int i = 0; i < pojas.size(); i++)
    // svaku tačku poredimo sa tackama iznad nje koje su u
    // pravougaoniku visine d
    for (int j = i+1; j < pojas.size() && pojas[j].y - pojas[i].y < d; j++) {
      // ako su dve tačke na rastojanju manjem od d, ažuriramo d
      double dij = rastojanje(pojas[i], pojas[j]);
      if (dij < d)
        d = dij;
    }

  // vraćamo pronađeno minimalno rastojanje
  return d;
}

// funkcija koja određuje rastojanje između dve najbliže tačke
// u datom skupu tačaka
double najblizeTacke(vector<Tacka>& tacke) {
  // sortiramo tačke na osnovu koordinate x
  sort(begin(tacke), end(tacke), porediX);
  // određujemo najbliži par tačaka rekukrzivnom funkcijom
  return najblizeTacke(tacke, 0, tacke.size() - 1);
}
```

Odredimo složenost prethodne funkcije. Algoritam se sastoji od dva rekurzivna poziva za dvostruko manju dimenziju niza tačaka i faze dobijanja krajnjeg rezultata na osnovu rezultata rekurzivnih poziva i dodatne analize tačaka u pojasu $[x-d, x+d]$. Već smo konstatovali da izdvajanje tačaka centralnog pojasa zahteva $O(n)$ memorije i vremena i da sortiranje tih tačaka po koordinati y zahteva dodatnih $O(n \log n)$ koraka. Ostaje još da se proceni složenost ugnežđenih petlji u kojima se porede tačke unutar pojasa. Iako deluje da je složenost kvadratna, elementarnim geometrijskim rezonovanjem dokazaćemo da je složenost tog koraka linearna tj. $O(n)$ i da se u svakom koraku spoljašnje petlje unutrašnja petlja može izvršiti samo veoma mali broj puta (dokazaćemo da je taj broj izvršavanja ograničen odozgo sa 7, mada je u praksi on često i dosta manji od toga i za nasumično generisane tačke ta petlja se najčešće izvršava 0, 1 ili eventualno dva puta).

Za svaku tačku A možemo konstruisati 8 kvadrata dimenzije $d/2$, kao što je prikazano na slici (kvadrati su upisani u pojas $[x-d, x+d]$, u dva reda od po četiri kvadrata i tačka A leži na donjoj ivici donjih kvadrata).

Najveće rastojanje između dve tačke unutar nekog kvadrata se postiže kada oni leže u njegovim naspramnim temenima, a pošto je dužina dijagonale kvadrata stranice $\frac{d}{2}$ jednaka $\frac{d\sqrt{2}}{2} \approx 0,70711 \cdot d$, rastojanje između svake dve tačke unutar istog kvadrata je strogo manje od d. Pošto svi kvadrati leže bilo potpuno sa leve strane vertikalne linije podele, bilo sa njene desne strane unutar svakog od kvadrata se može naći najviše jedna tačka našeg skupa (u suprotnom bi se bilo sa leve, bilo sa desne strane centralne linije podele nalazio par tačaka sa



Slika 2: Najbliži par tačaka

rastojanjem strogo manjim od d, što je kontradiktorno sa definicijom veličine d). To znači da se iznad tačke A može nalaziti najviše 7 tačaka koje pripadaju ostalim kvadratima (sama tačka A već pripada jednom od kvadrata) i da se sve ostale tačke koje su iznad A nalaze i iznad naših kvadrata, što znači da im je rastojanje od A sigurno veće od d (jer im je vertikalno rastojanje veće od d) i njih nije potrebno razmatrati.

Tačke koje su sa iste strane linije podele kao i tačka A možemo prosto preskočiti u telu unutrašnje petlje i tako uštediti na računanju njihovog rastojanja od tačke A, ali eksperimenti pokazuju da ta ušteda nije osetna. Druga mogućnost za implementaciju je da ne čuvamo sve tačke iz pojasa u istom skupu, već da ih podelimo u dva pojasa i da zatim da obradimo prvo sve tačke iz levog pojasa gledajući rastojanja u odnosu na naredne najviše 4 tačke iz desnog pojasa, a zatim da obradimo sve tačke iz desnog pojasa gledajući rastojanja u odnosu na najviše 4 tačke iz levog pojasa (jer u suprotnom pojasu postoji 4 kvadrata dimezije $d/2$, za koje smo dokazali da ne mogu da sadrže dve tačke istovremeno). Implementacija na taj način je malo komplikovanija, a eksperimenti ne ukazuju na značajne dobitke.

Dakle, nakon rekurzivnih poziva, za dobijanje konačnog rezultata je potrebno izvršiti dodatnih $O(n \log n)$ koraka i dekompozicija zadovoljava rekurentnu jednačinu $T(n) = 2T(n/2) + O(n \log n)$. Rešenje ove jednačine, na osnovu master teoreme, je $O(n(\log n)^2)$.

Složenost se može popraviti ako se sortiranje po koordinati y vrši istovremeno sa pronalaženjem najbližeg para tačaka, tj. ako se ojača induktivna hipoteza i ako se pretpostavi da će rekurzivni poziv vratiti rastojanje između najbliže dve tačke i ujedno sortirati date tačke po koordinati y. U koraku objedinjavanja dva sortirana niza objedinjujemo u jedan (uobičajenim algoritmom objedinjavanja, zasnovanom na tehnici dva pokazivača, koji je dostupan i pomoću bibliotečke funkcije `merge` i posao obavlja u linearnoj složenosti). Na taj način dobijamo algoritam koji zadovoljava jednačinu $T(n) = 2T(n/2) + O(n)$ i složenosti je

$O(n \log n)$. Naglasimo da ova optimizacija nije revolucionarna, ali može malo poboljšati efikasnost.

Na nivou implementacije, malo poboljšanje bismo mogli dobiti i tako što bismo izbegli alokacije pomoćnog vektora unutar rekurzivnih poziva i kod izmeniti tako da se u svakom rekurzivnom pozivu koristi isti, unapred alociran pomoćni vektor. Još jedna moguća optimizacija o kojoj bi se moglo razmisliti je smanjivanje broja operacija korenovanja.

```
// funkcija određuje najbliži par tačaka u delu niza [l, r]
// i sortira taj deo niza po koordinati y
double najblizeTacke(vector<Tacka>& tacke, int l, int r,
                     vector<Tacka>& pom) {
  // ako ima manje od 4 tačke, najmanje rastojanje pronalazimo
  // grubom silom
  if (r - l + 1 < 4) {
    // poredimo sve parove tačaka, tražeći minimalno rastojanje
    double d = numeric_limits<double>::max();
    for (int i = l; i < r; i++)
      for (int j = i+1; j <= r; j++) {
        double dij = rastojanje(tacke[i], tacke[j]);
        if (dij < d)
          d = dij;
      }
    // sortiramo niz po koordinati y
    sort(next(begin(tacke), l), next(begin(tacke), r+1), porediY);
    return d;
  }

  // pozicija središnje tačke
  int s = l + (r - l) / 2;
  // određujemo najmanje rastojanje tačaka koje su levo od
  // središnje tačke (uključujući i nju)
  // određujemo najmanje rastojanje tačaka koje su desno od
  // središnje tačke
  double d2 = najblizeTacke(tacke, s+1, r);
  // određujemo manje od ta dva rastojanja
  double d = min(d1, d2);

  // objedinjavamo dva sortirana niza tačka bibliotečkom funkcijom
  // u novi sortirani niz pom
  merge(next(begin(tacke), l), next(begin(tacke), s+1),
        next(begin(tacke), s+1), next(begin(tacke), r+1),
        begin(pom), porediY);
  // vraćamo sortirane tačke u originalni niz
  copy(begin(pom), end(pom), next(begin(tacke), l));

  // izdvajamo tačke iz pojasa [x-d, x+d] u vektor pom
  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)
      pom[k++] = tacke[i];

  // svaku tačku poredimo sa tačkama iznad nje koje su u
  // pravougaoniku visine d
  for (int j = i+1; j < pojas.size() && pojas[j].y - pojas[i].y < d; j++) {
    // ako su dve tačke na rastojanju manjem od d, ažuriramo d
    double dij = rastojanje(pojas[i], pojas[j]);
    if (dij < d)
      d = dij;
  }

  return d;
}

// funkcija određuje najbliži par datih tačaka, sortirajući usput
// niz po koordinati y (to je samo sporedni efekat nastao u cilju
// efikasne implementacije)
double najblizeTacke(vector<Tacka>& tacke) {
  vector<Tacka> pom(tacke.size());
  return najblizeTacke(tacke, 0, tacke.size() - 1, pom);
}
```