Algoritmi i strukture podataka
10. Čas — Dekompozicija

Sana Stojanović Đurđević

December 14, 2018

Pred vama je prateći materijal koji se može koristiti uz skriptu. Sastoji se od rešenja nekoliko zadataka koji su pređeni na vežbama i predavanjima.

1 Merge sort

Postoje bibliotečke funkcije koje se mogu koristiti za sortiranje objedinjavanjem: `merge` i `inplace_merge`.

merge funkcija objedinjuje dva sortirana niza (koji su zadati opsezima prva dva, odnosno druga dva iteratora) i rezultat smesta u treci sortirani niz zadat poslednjim (petim) argumentom.

Podrazumevano je da se elementi niza sortiraju u rastućem poretku (što može biti promenjeno po potrebi kao u primerima sa funkcijom `sort`). Podrazumevano je da su početni nizovi već sortirani i da se ne preklapaju.

`http://www.cplusplus.com/reference/algorithm/merge/`

```
// merge algorithm example
#include <iostream>     // std::cout
#include <algorithm>    // std::merge, std::sort
#include <vector>       // std::vector

using namespace std;

int main () {
  int first[] = {5,10,15,20,25};
  int second[] = {50,40,30,20,10};
  vector<int> v(10);

  sort (first,first+5);
  sort (second,second+5);
  merge (first,first+5,second,second+5,v.begin());

  cout << "The resulting vector contains:";
  for (vector<int>::iterator it=v.begin(); it!=v.end(); ++it)
    cout << ' ' << *it;
  cout << '\n';

  return 0;
}

Output:
The resulting vector contains: 5 10 10 15 20 20 25 30 40 50
```

inplace_merge funkcija objedinjava prvu i drugu polovinu datog niza koje su opisane početkom niza, sredinom niza i njegovim krajem. Pretpostavlja se da su polovine niza već sortirane. Takođe je podrazumevano da se koristi uobičajen način poređenja dva elementa niza, sem ako drugačije nije naglašeno.

Ova funkcija zadržava relativan odnos elemenata sa istom vrednošću, pri čemu će se elementi iz prve polovine naći pre elemenata iz druge polovine sa istom vrednošću.

`http://www.cplusplus.com/reference/algorithm/inplace_merge/`

```
// inplace_merge example
#include <iostream>     // cout
#include <algorithm>    // inplace_merge, sort, copy
#include <vector>       // vector

using namespace std;

int main () {
  int first[] = {5,10,15,20,25};
  int second[] = {50,40,30,20,10};
  vector<int> v(10);
  vector<int>::iterator it;

  sort (first,first+5);
  sort (second,second+5);

  it=copy (first, first+5, v.begin());
     copy (second,second+5,it);

  inplace_merge (v.begin(),v.begin()+5,v.end());

  cout << "The resulting vector contains:";
  for (it=v.begin(); it!=v.end(); ++it)
    cout << ' ' << *it;
  cout << '\n';

  return 0;
}

Output:
The resulting vector contains: 5 10 10 15 20 20 25 30 40 50
```

1.1 Sortiranje niza objedinjavanjem

Problem: Implementirati sortiranje niza objedinjavanjem sortiranih polovina.

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

using namespace std;

// 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 ucesljaj(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
    ucesljaj(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];
  }
}
```

```
// 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);
}
```

```
int main(){
  int n;
  cout << "Unesite dimenziju niza i elemente niza: ";
  cin >> n;
  vector<int> a(n);

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

  merge_sort(a);

  cout << "Sortirani niz: " << endl;
  for(int i=0; i<n; i++)
    cout << a[i] << " ";
  cout << endl;
}
```

1.2 Broj invezija u nizu

Problem: Odredi koliko postoji različitih parova elemenata u nizu tako da je prvi element tog para 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 rekurzivno odredi broj inverzija je da se niz sortira sortiranjem objedinjavanjem, prilagođenim tako da se broje inverzije.

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

using namespace std;

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);
}
```

```
int main(){
  int n;
  cout << "Unesite dimenziju niza i elemente niza: ";
  cin >> n;
  vector<int> a(n);

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

  cout << "Broj inverzija = " << broj_inverzija(a) << endl;
}
```

2 Quick-sort i quick-select

2.1 Quick-sort

Problem: Sortirati niz brojeva primenom algoritma brzog sortiranja.

U narednom kodu se za određivanje pivota koristi funcija `rand` koja vraća pseudo-slučajan ceo[1] broj iz intervala 0 i $RAND\_MAX$ (konstanta definisana bibliotekom). Nalazi se u zaglavlju `<cstdlib>`.

Ako želimo da dobijemo slučaju vrednost iz intervala $[l, d]$ potrebno je da koristimo ostatak pri deljenju pseudo-slučajnog broja: $l + rand()\%(d - l + 1)$.

`http://www.cplusplus.com/reference/cstdlib/rand/`

[1] Broj je pseudo-slučajni ako možemo da kažemo da se njegova vrednost smatra slučajnom, odnosno da je jednaka verovatnoća pojavljivanja bilo kog broja iz intervala 0 i $RAND\_MAX$.

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

using namespace std;

// 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);
}

int main(){
  int n;
  cout << "Unesite dimenziju niza, pa zatim elemente niza: " << endl;
  cin >> n;
  vector<int> a(n);
  for(int i=0; i<n; i++)
    cin >> a[i];

  quick_sort(a);

  cout << "Sortirani niz: ";
  for(int i=0; i<n; i++)
    cout << a[i] << " ";
  cout << endl;

  return 0;
}
```

2.2 Quick-select

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

Primetimo da je, umesto sortiranja celog niza, dovoljno implementirati modifikaciju 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).

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

using namespace std;

int random_value(int l, int d){
  return l + rand()%(d-l+1);
}
```

```
// 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)]);
    // particionisemo 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 vraamo 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;
    // zakljucno sa pivotom
    else if (k > j)
      l = j + 1;
    // pivot je tacno k-ti po redu
    else return a[k];
  }
}
```

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

```
int main(){
  int n;
  cout << "Unesite dimenziju niza, pa zatim elemente niza: " << endl;
  cin >> n;
  vector<int> a(n);
  for(int i=0; i<n; i++)
    cin >> a[i];

  cout << "Unesite k: " << endl;
  int k;
  cin >> k;

  int kti = ktiElement(a, k);

  cout << "Element od koga je tacno " << k;
  cout << " elemenata manje ili jednako je: " << kti << endl;
  return 0;
}
```

2.3 Zbir k najvećih elemenata niza

Problem U nizu od n elemenata izračunati zbir k najvećih elemenata niza.

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

using namespace std;

// QuickSelect - odredjujemo najvecih k elemenata niza a
// 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, int 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 vracamo rezultat
  int s = 0;
  for (int i = 0; i < k; i++)
    s += a[i];
  return s;
}
```

```
int main(){
  int n;
  cout << "Unesite dimenziju niza, pa zatim elemente niza: " << endl;
  cin >> n;
  vector<int> a(n);
  for(int i=0; i<n; i++)
    cin >> a[i];

  cout << "Unesite k: " << endl;
  int k;
  cin >> k;

  int zbir_k = zbirKNajvecih(a, k);

  cout << "Zbir " << k << " najvecih elemenata niza je: ";
  cout << zbirKNajvecih(a,k) << endl;
  return 0;
}
```

2.4 Zbir k najvećih elemenata niza, rešenje sa bibliotečkim funkcijama

U narednom rešenju se koristi nekoliko bibliotečkih funkcija. Program se mora prevesti sa opcijom `-std=c++11` zbog korišćenja funkcije `next`.

nth_element funkcija organizuje niz tako da je k-ti element na svom mestu (na kom bi se nalazio u sortiranom nizu) i da su svi elementi ispred njega manji od njega (ne obavezno sortirani). Korisnik može promeniti način poređenja navođenjem neke druge funkcije, kao što je urađeno u narednom primeru. Nalazi se u zaglavlju `<algorithm>`.

`http://www.cplusplus.com/reference/algorithm/nth_element/`

partial_sort funkcija obezbeđuje da je sortirano prvih k elemenata (i da su oni manji od elemenata iza k-tog, koji ne moraju biti sortirani). Nalazi se u zaglavlju `<algorithm>`.

`http://www.cplusplus.com/reference/algorithm/partial_sort/`

accumulate funkcija vraća "akumulirani izraz" vrednosti između dva iteratora, počevši sa početnom vrednošću koja se prosleđuje kao treći argument funkcije. Ako se funkcija poziva samo sa tri argumenta onda se te vrednosti sabiraju. Inace, može se zadati kao četvrti argument funkcija koja se koristi za akumulaciju.

`http://www.cplusplus.com/reference/numeric/accumulate/`

next funkcija vraća iterator koji je za n mesta pomeren u odnosu na inicijalni.

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

using namespace std;

int main(){
  int n;
  cout << "Unesite dimenziju niza, pa zatim elemente niza: " << endl;
  cin >> n;
  vector<int> a(n);
  for(int i=0; i<n; i++)
    cin >> a[i];

  cout << "Unesite k: " << endl;
  int k;
  cin >> k;

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

  return 0;
}
```

3 Karacubin algoritam množenja polinoma

Problem: Definisati funkciju koja množi dva polinoma predstavljena vektorima svojih koeficijenata. Jednostavnosti radi pretpostaviti da su dužine vektora stepeni dvojke.

Prilikom množenja polinoma $a+bx$ i $c+dx$, potrebno je izračunati polinom $ac+(ad+bc)x+bd$, 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 $(a+b)(c+d)$.

3.1 Rešenje sa pomoćnim vektorima

http://www.cplusplus.com/reference/algorithm/copy_n/

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

using namespace std;

// funkcija mnozi dva polinoma p1*p2
vector<double> proizvod_polinoma(const vector<double>& p1,
                                 const vector<double>& p2) {
  int n = p1.size();
  vector<double> proizvod(2*n, 0);
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
      proizvod[i+j] += p1[i] * p2[j];
  return proizvod;
}

// funkcija stampa polinom
void odstampaj_polinom(vector<double> p){
  int n = p.size();
  cout << p[0];
  for(int i=1; i<n; i++)
    cout << " + " << p[i] << " * x^" << i;
  cout << endl;
}
```

U narednom kodu koristi se funkcija `copy_n` koja je dostupna od standarda C++11.

```
// funkcija mnozi dva polinoma p1*p2 sa po n = 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 mnozimo
  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));
```

```
  // Vazi:
  //   (a+bx)*(c+dx) = a*c + ((a+b)*(c+d) - a*c - b*d)*x + b*d*x^2
  // rekurzivno racunamo a*c i b*d
  vector<double> ac = karacuba(a, c);
  vector<double> bd = karacuba(b, d);

  // izracunavamo a+b (koristimo pomocni vektor a)
  for (int i = 0; i < n/2; i++)
    a[i] += b[i];

  // izracunavamo c+d (koristimo pomocni vektor b)
  for (int i = 0; i < n/2; i++)
    c[i] += d[i];

  // izracunavamo (a+b)*(c+d)
  vector<double> abcd = karacuba(a, c);

  // izracunavamo (a+b)*(c+d) - a*c - b*d
  for (int i = 0; i < n; i++)
    abcd[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] += abcd[i];
    proizvod[i] += ac[i];
  }

  // vracamo rezultat
  return proizvod;
}
```

```
int main(){
  int n;
  cout << "Unesite vrednost n (stepen dvojke):";
  cin >> n;
  cout << "Unesite koeficijente prvog polinoma ";
  cout << "(pocevsi od koeficijenta uz najmanji stepen):" << endl;
  vector<double> p1(n);
  for(int i=0; i<n; i++)
    cin >> p1[i];

  cout << "Unesite koeficijente drugog polinoma ";
  cout << "(pocevsi od koeficijenta uz najmanji stepen):" << endl;
  vector<double> p2(n);
  for(int i=0; i<n; i++)
    cin >> p2[i];

  cout << "Proizvod polinoma: " << endl;
  odstampaj_polinom(p1);
  odstampaj_polinom(p2);
  cout << "je: " << endl;

  vector<double> proizvod(2*n-1);
  proizvod = proizvod_polinoma(p1,p2);
  odstampaj_polinom(proizvod);

  vector<double> k_proizvod(2*n-1);
  k_proizvod = karacuba(p1,p2);
  cout << "Karacuba: " << endl;
  odstampaj_polinom(k_proizvod);

  return 0;
}
```

3.2 Rešenje koje troši manje memorije

Problem koji se javlja sa prethodnom implementacijom 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 pomonu memoriju alocirati samo jednom i onda tokom rekurzije koristiti stalno isti pomoćni memorijski prostor. Veličina potrebne pomoćne memorije je $2n$ (dva puta po $n/2$ da se smeste polinomi $a+b$ i $c+d$ i još n da se smesti njihov proizvod). Dodatna optimizacija se može dobiti ako 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 $n \leq 4$.

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

using namespace std;

// funkcija stampa polinom
void odstampaj_polinom(vector<double> p){
  int n = p.size();
  cout << p[0];
  for(int i=1; i<n; i++)
    cout << " + " << p[i] << " * x^" << i;
  cout << endl;
}
```

```
// mnozimo polinome ciji su koeficijenti smesteni u vektorima
// p1[start1, start1+n) i p2[start2, start2+n)
// i rezultat smestamo u vektor
// proizvod[start_proizvod, start_proizvod + 2n),
// koristeci pomocni memorijski prostor u vektoru
// pom[start_pom, start_pom + 2n)
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) {
    // klasicni algoritam mnozenja
    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;
  }
```

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

  // Izracunavamo rekurzivno a*c i smestamo ga u levu polovinu
  // proizvoda
  karacuba(n / 2, p1, start1, p2, start2,
   proizvod, start_proizvod, pom, start_pom);

  // Izracunavamo rekurzivno b*d i smestamo ga u desnu polovinu
  // proizvoda
  karacuba(n / 2, p1, start1 + n/2, p2, start2 + n/2,
   proizvod, start_proizvod + n, pom, start_pom);

  // Izracunavamo a+b i smestamo ga u pomocni vektor (na pocetak)
  for(int i = 0; i < n/2; i++)
    pom[start_pom + i] =
      p1[start1 + i] + p1[start1 + n/2 + i];

  // Izracunavamo c+d i smestamo ga u pomocni 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 izracunavamo (a+b)*(c+d) i smestamo ga
  // u pomocni 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);

  // Izracunavamo (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 mnozi 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);
  // pomocni memorijski prostor potreban za realizaciju algoritma
  vector<double> pom(2 * n);
  // vrsimo mnozenje
  karacuba(n, p1, 0, p2, 0, proizvod, 0, pom, 0);
  // vracamo proizvod
  return proizvod;
}
```

```
int main(){
  int n;
  cout << "Unesite vrednost n (stepen dvojke):";
  cin >> n;
  cout << "Unesite koeficijente prvog polinoma ";
  cout << "(pocevsi od koeficijenta uz najmanji stepen):" << endl;
  vector<double> p1(n);
  for(int i=0; i<n; i++)
    cin >> p1[i];

  cout << "Unesite koeficijente drugog polinoma ";
  cout << "(pocevsi od koeficijenta uz najmanji stepen):" << endl;
  vector<double> p2(n);
  for(int i=0; i<n; i++)
    cin >> p2[i];

  vector<double> k_proizvod(2*n-1);
  k_proizvod = karacuba(p1,p2);
  cout << "Karacuba: " << endl;
  odstampaj_polinom(k_proizvod);

  return 0;
}
```

4 Brza Furijeova transformacija

U narednom kodu se koristi nekoliko pogodnosti jezika C++.

complex U jeziku C++ kompleksne brojeve imamo na raspolaganju u obliku tipova `complex<double>` i `complex<float>` (zapis u dvostrukoj i jednostrukoj tačnosti). Potrebno je uključiti zaglavlje `<complex>`.

Imaginaran broj i onda možemo zadati na sledeći način: `complex<double> ii(0.0,1.0);`

real, imag funkcije vraćaju realan i imaginaran deo kompleksnog broja
`http://www.cplusplus.com/reference/complex/complex/real/`
`http://www.cplusplus.com/reference/complex/complex/imag/`

exp funkcija računa vrednost e^x za zadatu vrednost x. Nalazi se u zaglavlju `<math>`.
`http://www.cplusplus.com/reference/complex/exp/`

```
#include <iostream>
#include <vector>
#include <complex>
#include <math.h>

using namespace std;

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) {

  complex<double> ii(0.0,1.0);
  // 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)*ii);

    // 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;
}
```

```
int main(){

  int n;
  cout << "Unesite broj koeficijenata polinoma" << endl;
  cout << "(stepen polinoma je n-1), koji je stepen dvojke" << endl;
  cout << "i koeficijente polinoma: " << endl;
  cin >> n;
  ComplexVector a(n);
  for(int i=0; i<n; i++)
    cin >> a[i];
  ComplexVector rezultat(n);

  rezultat = fft(a);

  for(int i=0; i<n; i++)
    cout << rezultat[i] << endl;

  return 0;
}
```