1. Napisati algoritam za CRCW/EREW PRAM računalo koji će za zadano polje P[] provjeriti ima li u polju elemenata jednakih vrijednosti (duplikata). Za polje od n elemenata na raspolaganju je n procesora. Rezultat mora biti zapisan u jednoj izlaznoj varijabli. Ocijeniti složenost algoritma. Složenost O(n)

CRCW:

Uspoređujem prvi s drugim, drugi s trećim… Drugi korak uspoređujem prvi s trećim, drugi s četvrtim…

```
Rez = 0;
FOR(i=1; i<n; i++){
        PARALELNO(j=0; j<n-i; j++){
                IF(P[j]==P[j+i])
                        Rez=1;
        }
}
```

EREW:

Kopiramo polje P u polje K u O(1). Stvaramo polje DUPLIKATI. Na indexu „i“ u polju DUPLIKATI piše 1 ukoliko je element u polju P indeksiran sa „i“ duplikat, inače piše 0.

```
PARALELNO(i=0; i<n; i++){
        K[i]=P[i];
        DUPLIKATI[i]=0;
}
FOR(i=1; i<n; i++){
        PARALELNO(j=0; j<n-i; j++){
                IF(P[j] == K[j+i])
                        DUPLIKATI[j]=1;
        }
}
Rez=OR_REDZCE(DUPLIKATI[]);
```

2. Napisati algoritam za CRCW/EREW PRAM računalo koji će za zadano polje P[] odrediti broj različitih vrijednosti elemenata polja. Npr. za polje [1, 2, 1, 3, 4, 2, 5, 1] rezultat iznosi 5. Za polje od n elemenata na raspolaganju je n procesora. Rezultat mora biti zapisan u jednoj izlaznoj varijabli. Ocijeniti složenost algoritma. Složenost O(n)

CRCW:

Napravi dodatno polje Jedinstveni veličine polja P gdje su svi elementi 1. Prvi korak petlje, procesor uspoređuje svoju vrijednost s prvom u polju. Ako su jednake, na lokaciji Jedinstveni[index] upisati 0. Na kraju napraviti +_reduce.

```
PARALELNO(i=0; i<n; i++){
        JEDINSTVENI[i]=1;
}
FOR(i=0; i<n; i++){
        PARALELNO(j=i+1; j<n; j++){
                IF(P[i]==P[j])
                        JEDINSTVENI[j]=0;
        }
}
Rez = +_reduce(JEDINSTVENI[]);
```

EREW:

Polje P[] se kopira u polje K[] (O(1)). Slično kao u CRCW mogu se raditi usporedbe, te ako nešto nije jedinstveno zapisuje se 0 u UNIQUE[] na odgovarajućem indeksu. Konačno, napravimo +_reduce.

```
paralelno( i = 0; i < n; i++) {
        UNIQUE[ i ] = 1;
```

```
        K [ i ] = P [ i ];
}
for ( i = 1; i < n; i++) {
        paralelno( j = 0; j < n - i; j++) {
                if( P [ j ] == K[ j + i ] )
                        UNIQUE[ j + i] = 0;
        }
}
Rez = +_reduce(UNIQUE []);
```

3. Napišite algoritam složenosti log(n) za EREW PRAM računalo koji će za zadano polje A[] s n elemenata odrediti je li uzlazno sortirano. Na raspolaganju je funkcija +_reduce(polje[]) koja provodi operaciju zbrajanja. Složenost O(logn)

```
paralelno( i = 0; i < n; i++) {
        K [ i ] = A [ i ];
        R [ i ] = 1;
} // O (1)
paralelno(i = 0; i < n - 1; i++) {
        if( A [ i ] > K [ i + 1])
                R [ i ] = 0;
} // O (1)
rezultat =  +_reduce(R[]); // O ( log n );

if( rezultat == n)
        //Polje je sortirano
```

4. Napisati algoritam za EREW PRAM računalo složenosti manje od $O(n)$koji će za dva ulazna niza znakova A[] i B[] duljine n odrediti koji je prvi po abecedi (program treba ispisati "A", "B" ili "jednaki"). Na raspolaganju je funkcija reduciranja koja provodi proizvoljnu binarnu operaciju nad elementima polja. Ocijeniti složenost algoritma. Proizvoljnu operaciju možete definirati programski.

Složenost O(logn)

```
nađi(a,b) {
        if( a == 0)
                return b;
        else
                return a;
}
paralelno ( i = 0; i < n; i++) {
        if( A[ i ] < B [ i ])
                R[ i ] = - 1;
        if( A[ i ] < B [ i ])
                R[ i ] = - 1;
        else if( A [ i ] > B [ i ])
                R [ i ] = 1;
        else
                R[ i ] = 0;
}
odg  = nađi_reduciraj( R [] );
if( odg == -1)
        odg = “A”;
else if(odg == 1)
```

```
        odg = “B”;
else
        odg = “jednak”;
```

5. Napišite algoritam za EREW PRAM računalo složenosti manje od O(n^2) koji će za matricu M[n][n] odrediti predstavlja li permutacijsku matricu (u svakom retku i stupcu je točno jedna jedinica, a ostali elementi su nule). Program treba ispisati "da" ili "ne". Na raspolaganju je n procesora i funkcija reduciranja (O(log n)) koja provodi proizvoljnu binarnu operaciju nad elementima niza. Pretpostavite da su elementi matrice bilo koje cjelobrojne vrijednosti (mogu biti različiti od nula i jedan!), te da se i-ti redak odnosno stupac matrice može dobiti kao R[i] odnosno S[i]. Ocijenite složenost algoritma. Proizvoljnu operaciju možete definirati programski (u obliku funkcije).
Složenost O(n*logn)

```
PERM = 1
ZA (i=0; i<n; i++)
        PARALELNO(j=0 do n-1)
                Ako(R[i][j] != {0,1} || S[i][j] != {0,1})
                        Nije_perm[j]=1;
        Ako(OR_reduce(nije_perm[])==1)
                PERM=0;
        AKO (+_reduce(R[i]) != 1 ILI +_reduce(S[i]) != 1)
                PERM = 0
AKO (PERM == 1)
        ISPIŠI "Da"
INAČE
        ISPIŠI "Ne"
```

6. Napisati algoritam za CRCW PRAM računalo koji će, ispisivanjem "DA" ili "NE", za zadano polje P[] sa n elemenata odrediti predstavlja li permutacijski vektor (permutacija skupa {1, 2, ..., n}). Elementi polja mogu poprimiti samo cjelobrojne vrijednosti. Primjerice, (3, 1, 2) je permutacijski vektor dok (2, 4, 3), (1, 1, 2) i (4, 2, 1) nisu. Na raspolaganju je funkcija reduciranja koja provodi proizvoljnu binarnu operaciju nad elementima polja. Ocijeniti složenost algoritma.
Složenost O(n)

```
PARALELNO(i=0 do n-1)
        Broj[i]=0;
        ZA(j=0 do n-1)
                AKO(P[j]==i)
                        Broj[i]++;
Permutacija = AND_REDUCE(Broj[]);
AKO(Permutacija = 1)
        ISPIŠI Da
INAČE
        ISPIŠI Ne
```

7. Napisati algoritam za EREW PRAM računalo koji će za zadano polje P[] sa n elemenata odrediti predstavlja li neki redak permutacijske matrice (jedan element 1, svi ostali 0). O vrijednostima elemenata polja se ništa ne pretpostavlja (mogu biti bilo koje vrijednosti). Na raspolaganju je funkcija *_reduce(polje[]) koja provodi proizvoljnu binarnu operaciju nad elementima polja. Ocijeniti složenost algoritma.
Složenost O(logn)

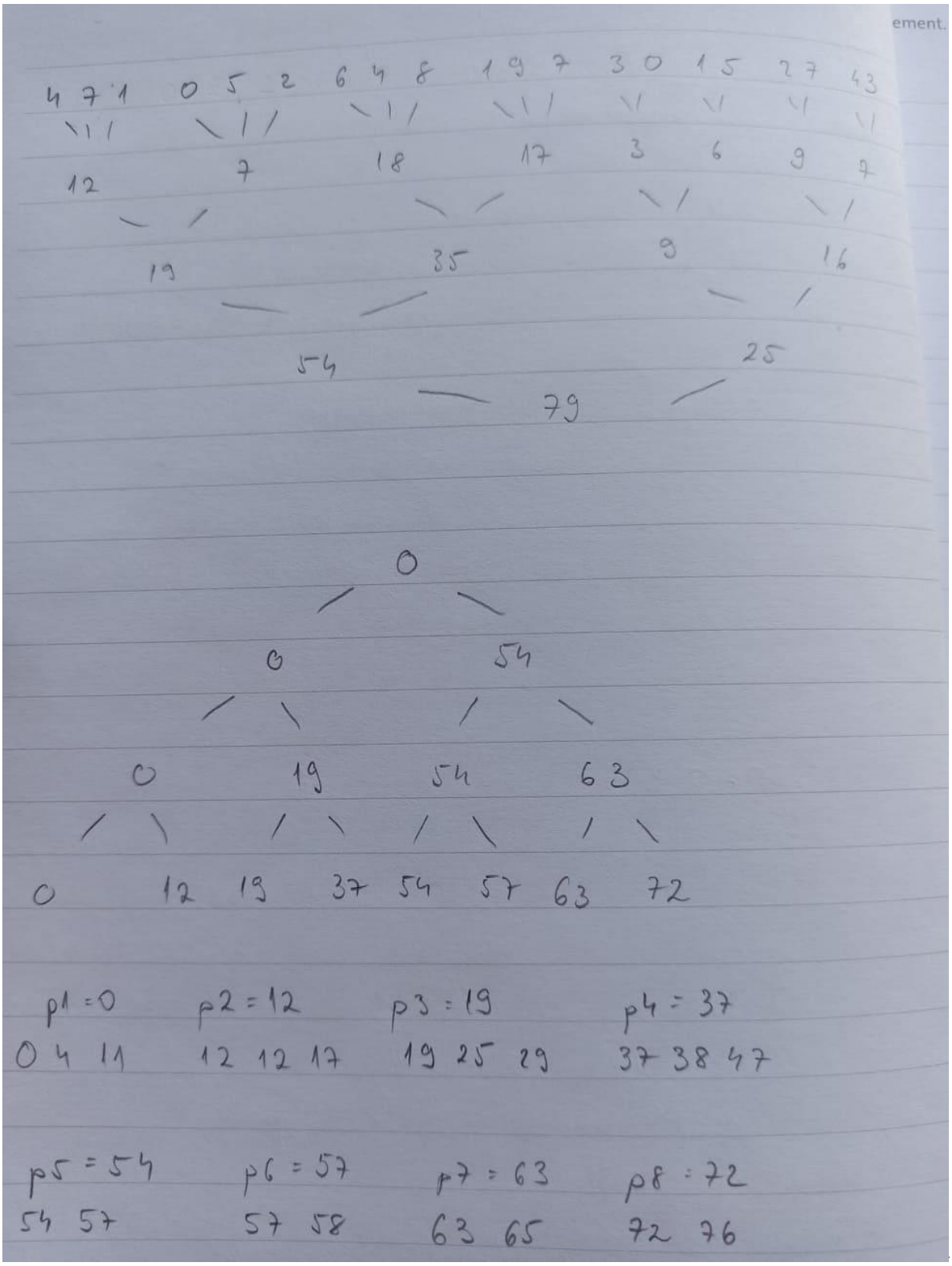
```
NULE = [];
JEDINICE = [];
PARALELNO (i = 0 DO n-1)
        AKO (P[I] == 0)
                NULE[i] = 1;
        INAČE
                NULE[i] = 0;
        AKO (P[i] == 1)
                JEDINICE[i] = 1;
        INAČE
                JEDINICE[i] = 0;
R0 = +_REDUCE(NULE[]);
R1 = +_REDUCE(JEDINICE[]);
AKO (R0 == n-1 && R1 == 1)
        ISPIŠI “DA”;
INAČE
        ISPIŠI “NE”;
```

8. U ostvarenju igre 4 u nizu prazni elementi ploče su označeni nulom, a pozicije s igračevim žetonom jedinicom (ne postoje elementi drugih vrijednosti). Napišite algoritam za EREW PRAM računalo koji za zadani (jednodimenzijski) niz elemenata ploče P[] duljine n otkriva postoji li u njemu 4 igračeva žetona u nizu (ispisuje DA ili NE). Na raspolaganju su scan i reduce funkcije za proizvoljne operacije. Netrivijalne operacije (npr. one koje uključuju grananja) potrebno je definirati algoritamski.

Složenost O(logn)

```
PARALELNO(i=0 do n-4)
        NIZ[i]=P[i];
        ZA(j=1 do 3)
                NIZ[i] += P[i+j];
M = MAX_REDUCE(NIZ[]);
AKO(M==4)
        ISPIŠI Da;
INAČE
        ISPIŠI Ne;
```

9. Provesti *+_prescan* algoritam na zadanom polju duljine n = 20 elemenata i na p = 8 procesora. Označiti podjelu elemenata po procesorima i tablično napisati izvedbu algoritma. Ulazno polje je A[] = [4 7 1 0 5 2 6 4 8 1 9 7 3 0 1 5 2 7 4 3].



10. Napišite algoritam za EREW PRAM računalo koji za zadani niz cjelobrojnih vrijednosti duljine *n* uz najviše *n* procesora otkriva duljinu najduljeg neprekinutog podniza elemenata s jednakom vrijednošću (ispisati duljinu podniza). Na raspolaganju su *scan* i *reduce* funkcije za proizvoljne operacije. Netrivijalne operacije (npr. one koje uključuju granja) potrebno je definirati algoritamski. Ocijenite složenost algoritma. Složenost O(n)

```
// P[n] ulazno polje
isti[N-1] = {}
PARALELNO(i=0 DO n-2)
  isti[i] = 0
PARALELNO(i=1 DO n-1)
  AKO(P[i-1] == P[i])
    isti[i-1] = 1
duljina = +_scan(isti[])
PARALLENO(i=0 DO n-2)
  AKO(isti[i] == 1)
    duljina[i] = 0
duljina = max_scan(duljina[])
PARALLENO(i=1 DO n-2)
  AKO(isti[i] == 0)
    isti[i] = duljina[i-1] - duljina[i]
duljina = +_scan(isti[])
najdulji = 1 + max_reduce(duljina[])
```

MPI

1. Korištenjem MPI funkcija Send i Recv (skraćena sintaksa) napišite niz instrukcija koji će sve elemente zadane kružne liste postaviti na srednju vrijednost toga i dvaju susjednih elemenata (indeksi i, i+1, i-1; posljednji element povezan je s prvim i obrnuto). Svaki MPI proces ima u lokalnoj memoriji samo jedan element liste koji je realna vrijednost. Broj procesa je N, svaki proces ima redni broj ID. *Program treba jamčiti ispravnost rada bez obzira na veličinu poruka* (ne smije doći do potpunog zastoja zbog redoslijeda slanja i primanja)!

```
double my_val_copy = my_val;
MPI_Send(&my_val,1,MPI_DOUBLE,(m_rank+w_size-1)%w_size,0,MPI_COMM_WORLD);
MPI_Recv(&m_buff,1,MPI_DOUBLE,(m_rank+w_size-1)%w_size,0,MPI_COMM_WORLD,&ls);
my_val_copy += m_buff;
MPI_Send(&my_val,1,MPI_DOUBLE,(m_rank+1)%w_size,0,MPI_COMM_WORLD);
MPI_Recv(&m_buff,1,MPI_DOUBLE,(m_rank+1)%w_size,0,MPI_COMM_WORLD,&ls);
my_val_copy += m_buff;
my_val_copy /= 3.0;
//0 proces mora prvo recv onda send
```

2. U jednom trenutku rada paralelnog programa u *n* procesora se nalaze neki podaci. Potrebno je odrediti najveći element od svih *n* podataka i tu informaciju (vrijednost najvećega) proslijediti svim procesorima. Napisati algoritam koji će obaviti taj zadatak pomoću MPI funkcija MPI_Send i MPI_Recv. Uputa: slanje i primanje poruka obaviti u obliku lanca u dva prolaza (s lijeva na desno te potom s desna na lijevo po svim procesorima). Kod poziva MPI funkcija navesti samo 'bitne' parametre (npr. MPI_Send(&varijabla,_,_,odrediste,_,_);).

```
If (my_rank == 0)
        MPI_Send(&my_number,_,_,1,_,_); // proces 0 šalje procesu 1
else if(my_rank > 0 && my_rank < world_size - 1) {
        MPI_Recv(&rec_buff,_,_,my_rank-1,_,_,_); // prvo primi od procesa lijevo od sebe
        my_number = rec_buff >= my_number ? rec_buff : my_number;
        MPI_Send(&my_number,_,_,my_rank+1,_,_); //šalji jednom desno od sebe.
}
```

```
else
        MPI_Recv(&rec_buff,_,_,my_rank-1,_,_,_);
        my_number = rec_buff >= my_number ? rec_buff : my_number;
if(my_rank == world_size - 1)
        MPI_Send(&my_number,_,_,my_rank-1,_,_);
else if(my_rank > 0 && my_rank < world_size - 1) {
        MPI_Recv(&rec_buff,_,_,my_rank+1,_,_,_); // Prvo primi od procesa desno od sebe.
        my_number = rec_buff >= my_number ? rec_buff : my_number;
        MPI_Send(&my_number,_,_,my_rank-1,_,_);
}
else {
        MPI_Recv(&rec_buff,_,_,my_rank+1,_,_,_);
        my_number = rec_buff >= my_number ? rec_buff : my_number;
}
```

3. Korištenjem MPI funkcija *Send* i *Recv* (skraćena sintaksa) napisati odsječak programa (proizvoljne složenosti) koji će za *N* procesa ostvariti funkciju MPI_Barrier, tj. postići da svi procesi moraju doći do istog odsječka prije nego bilo koji proces može nastaviti s izvođenjem. (U svakom procesu varijabla *ID* je indeks, a varijabla *N* ukupni broj procesa.)

1. Kada se dođe do trenutka gdje se želi ostvariti barijera potrebno je svim ostalim procesima poslati porijere. (n - 1 poruka)
2. Svaki proces treba primiti (n -1 poruku) od ruku da se došlo do badrugih procesa da su stigli do barijere. To se ponavlja n puta, pa se ukupno prima n * (n - 1) poruka.

Sveukupno je ovo (n + 1) * (n - 1) poruka. Otprilike n^2 poruka.

4. Zadan je MPI program (na slici desno). Svi procesi imaju lokalne varijable *a*, *b* i *c*, a *ID* je indeks pojedinog procesa. Koje vrijednosti će imati varijabla *c* za svaki proces na kraju izvođenja? Navedite sve mogućnosti i skicirajte redoslijed izvođenja MPI operacija za svaki proces.

```
Proces 1, ID = 1
MPI_Send(&ID, _, _, 2, _, _);
MPI_Recv(&a, _, _, MPI_ANY_SOURCE, _, _);
MPI_Send(&a, _, _, 3, _, _);
MPI_Recv(&b, _, _, MPI_ANY_SOURCE, _, _);
c = 2*a + b;
// Proces 2, ID = 2
MPI_Recv(&a, _, _, MPI_ANY_SOURCE, _, _);
MPI_Recv(&b, _, _, MPI_ANY_SOURCE, _, _);
c = 2*a + b;
MPI_Send(&c, _, _, 1, _, _);
MPI_Send(&ID, _, _, 3, _, _);
// Proces 3, ID = 3
MPI_Send(&ID, _, _, 2, _, _);
MPI_Recv(&a, _, _, MPI_ANY_SOURCE, _, _);
MPI_Recv(&b, _, _, MPI_ANY_SOURCE, _, _);
MPI_Send(&a, _, _, 1, _, _);
c = 2*a + b;
```

Ovdje 4 mogućnosti ? Send procesa 1 dode prvi do procesa 2 ili Send procesa 3 dode prvi do procesa 2. I za svaki od ta dva slucaja ili Send broj 2 u procesu 1 dode prvi do procesa 3 ili Send broj 2 procesa 2 dode prvi do procesa 3.

5. U MPI programu svaki proces ima lokalnu vrijednost u varijabli *x*. Korištenjem MPI funkcija *Send* i *Recv* (skraćena sintaksa) napisati odsječak programa logaritamske

složenosti (po pitanju broja poslanih poruka) koji će za *N* procesa izračunati minimum svih lokalnih vrijednosti, tako da svi procesi znaju rezultat. (U svakom procesu varijabla *ID* je indeks, a varijabla *N* ukupni broj procesa.)

Koristi se struktura hiperkocke

```
ZA (i = 0; i < logN; ++i) {
        DEST_ID = ID XOR 2^ i;
        SEND(x, DEST_ID);
        RECV(njegov_x, DEST_ID);
        If (njegov_x < x) {
                x = njegov_x;
        }
}
```

6. U MPI programu u nekom trenutku pojavio se promatrani događaj unutar jednog MPI procesa. Svi procesi znaju da se događaj pojavio, ali nijedan proces (osim dotičnog, izvorišnog procesa) ne zna unutar kojeg procesa se pojavio događaj (tj. koji je proces izvorišni). Korištenjem MPI funkcija *Send* i *Recv* (skraćena sintaksa) napisati odsječak programa logaritamske složenosti (po pitanju broja poslanih poruka) koji će za *N* procesa omogućiti da svi procesi saznaju indeks izvorišnog procesa. (U svakom procesu varijabla *ID* je indeks, a varijabla *N* ukupni broj procesa)

```
Izvor = -1;
If (događaj) izvor = ID;
ZA (i = 0; i < logN; ++i) {
        DEST_ID = ID XOR 2^ i;
        SEND(izvor, DEST_ID);
        RECV(njegov_izvor, DEST_ID);
        If (njegov_izvor > -1) {
                izvor = njegov_izvor;
        }
}
```

7. U MPI programu u nekom trenutku svih *N* procesa treba obaviti kritični odsječak. Svaki proces zna svoj redni broj ulaska u K.O., ali ne zna redne brojeve ostalih procesa. Korištenjem MPI funkcija *Send* i *Recv* (skraćena sintaksa) napisati odsječak programa logaritamske složenosti (po pitanju broja poslanih poruka) koji će omogućiti da svaki proces sazna indeks svog neposrednog prethodnika i sljedbenika (pozivanje K.O. nije potrebno prikazati). U svakom procesu varijabla *ID* je indeks procesa, a varijabla RBR redni broj ulaska u K.O.

```
lista = int[N]
za(i = 0 do n-1):
   lista[i] = -1
lista[RBR-1] = ID
za(i = 0 do log n):
   dest_ID = ID XOR 2^i
   send(lista, dest_ID)
   recv(druga_lista, dest_ID)
   za(i = 0 do n-1):
      ako(druga_lista[i] != -1 && lista[i] == -1):
         lista[i] = druga_lista[i]
prethodnik = RBR != 1 ? lista[RBR-1-1] : -1
sljedbenik = RBR != N ? lista[RBR-1+1] : -1
```