Skripta iz predmeta Paralelni algoritmi

Specijalističke studije

Prirodno-matematički fakultet UCG u Podgorici

- **Kompajler ili operativni sistem je odgovoran za paralelizaciju** ako nije eksplicitno naglašeno da li je paralelizam potreban.

- **Broj procesora u paralelnim računarskim sistemima** je obično stepen dvojke.

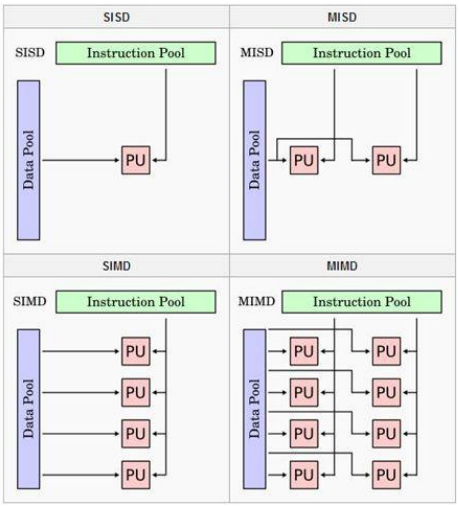
- ***Select /*+ parallel (4,t)*/ * from tabela t*** – SQL pseudo-instrukcija kojom vršimo selekciju podataka iz tabele t koristeći 4 procesora

- **Atributi paralelnog modela izračunavanja:**
  - ***osnovne (primitivne) jedinice računara*** – osnovne aktivnosti računara, tj. elementarni tipovi podataka i skup instrukcija koje računar može da izračuna ili izvrši.
  - ***mehanizam podataka***, definiše gdje čuvamo podatke i kako ćemo da pristupamo tim podacima
  - ***kontrolni mehanizam*** ili ***mehanizam upravljanja***, rešava problem podjele problema na manje i jednostavnije probleme (primitivne jedinice) i načina organizacije toka radnji tih primitivnih jedinica.
  - ***način komunikacije*** – ako imamo više uređaja u našem modelu izračunavanja, treba da odredimo način komunikacije između njih
  - ***mehanizam sinhronizacije*** – omogućava da pravi podaci budu obrađeni u pravo vrijeme

- **Fon Nojmanovi principi:**
  - podaci i programi se čuvaju u memoriji,
  - instrukcija u sebi nosi informaciju o operandima
  - instrukcije se izvršavaju sekvencijalno, sve dok se ne dođe do instrukcije skoka koja mijenja tok izvršavanja. Kod podjele problema, bitno je da svaki dio ima instrukcije koje se izvršavaju jedna za drugom.
  - reprezentacija instrukcija i podataka je ista, jer sama instrukcija u nekom trenutku može da posluži kao podatak.

- **Flinova klasifikacija (podjela prema broju tokova instrukcija i tokova podataka):**
  - ***SISD*** – jedan tok instrukcija i jedan tok podataka. Ovakvu situaciju srećemo kod klasičnog jednoprocesorskog računara.
  - ***SIMD*** – jedan tok instrukcija i više tokova podataka. Ovo je primjer arhitekture vektorskih računara.
  - ***MISD*** – više tokova instrukcija i jedan tok podataka. Kao tipičan primjer ove grupe možemo navesti *radnu liniju.* Koristimo ga kod protočne obrade.
  - ***MIMD*** – više tokova instrukcija i više tokova podataka. Ovo je najinteresantnija klasa i najrasprostranjeniji model.

Flynn's Classification



- **Način komunikacije paralelnih računara (tipovi 2 i 4 u Flinovoj klasifikaciji):**
  - ***korišćenje zajedničkog adresnog prostora***, a kao specifičan slučaj ove grupacije imamo ***korišćenje zajedničke memorije***.
  - ***komunikacija preko poruka*** ili tzv. ***mrežni model***.

  Kod prvog načina možemo imati više procesora koji pristupaju adresnim lokacijama u jedinstvenoj memoriji ili sistem u kome je memorija izdijeljena na više zasebnih cjelina tako da svaki procesor može pristupiti svakoj memoriji putem nekog međusloja. Ovdje se javlja problem konkurentnog upisa (više procesora pokušava istovremeno promijeniti stanje neke memorijske lokacije)

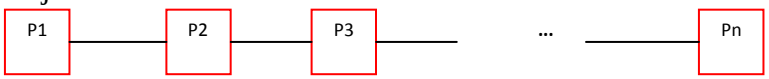
- **Klasifikacija prema tome da li je dozvoljeno istovremeno upisivanje/čitanje podataka u/iz zajedničke memorije od strane više procesora:**
  - **EREW**
  - **ERCW**
  - **CREW**
  - **CRCW**

  gdje E označava ekskluzivnost, C konkurentnost, R mogućnost čitanja, a W mogućnost upisa .

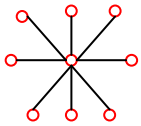
- **Modeli kod kojih je dozvoljen istovremeni upis u zajedničku memoriju:**
  - Dozvoljavamo upis, ako **svi procesori žele da upišu istu vrijednost**
  - Dozvoljavamo **upis po unaprijed zadatom prioritetu** (npr. procesor sa najmanjim indeksom ima najveći prioritet, pa on vrši upis)
  - **Procesor** kome ćemo dozvoliti upis **se bira na osnovu neke funkcije svih argumenata** koje treba upisati (suma, maksimum, minimum, prosjek, ...) čiji ishod ne smije da zavisi od rasporeda argumenata u funkciji; ova funkcija mora biti simetrična u odnosu na svoje argumente
  - **Slučajno biramo procesor** kome dozvoljavamo upis
    Kod ovih modela mora postojati kontrolni mehanizam koji vrši konačnu provjeru prije upisa (npr. kod prvog modela provjerava da li su sve vrijednosti iste, ako nijesu, u nekom koraku prekida upis i javlja grešku)

- **Kod mrežnog modela procesore povezujemo preko komunikacionih kanala.** Koriste se operacije **SEND za slanje poruke od jednog ka drugom procesoru** i **RECEIVE za prijem poruke i prepoznavanje pošiljaoca**.

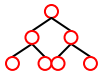
- **Klasifikacija kod mrežnog modela:**
  - **Svaki procesor povezan sa svakim**
  - **Linearno (ulančano) povezani procesori** – svaki procesor komunicira sa susjednim

  - **Procesori povezani u obliku prstena** (linearni model kod koga je prvi procesor povezan sa poslednjim procesorom)
  - **Procesori povezani u obliku zvijezde**

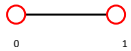
    
  - **Procesori povezani u obliku rešetke**

    
  - **Procesori povezani u obliku stabla**

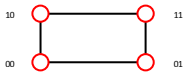
    
  - **Procesori povezani u obliku hiperkocke**
  - *i njima slični modeli kao i modeli nastali kombinovanjem pomenutih (mješoviti)*

- **Hiperkocka**
  - Hiperkocka *n*-tog nivoa ima 2^n procesora. Hiperkocku *n*-tog nivoa gradimo povezivanjem dvije hiperkocke *n-1*-vog nivoa. Povezivanje se vrši tako što prvo numerišemo procesore brojevima u binarnom brojnom sistemu i potom spajamo one procesore čiji se broj međusobno razlikuje u tačno jednoj cifri (npr. 10 sa 11).

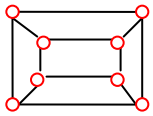
- **Jednodimenzionalna hiperkocka**



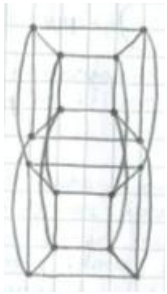
- **Dvodimenzionalna hiperkocka**



- **Trodimenzionalna hiperkocka**



- **Četvorodimenzionalna hiperkocka**



- **Transpjuter**
  - Transpjuter se definiše kao procesor koji sa sve svoje četiri strane ima četiri veze za spajanje sa drugim procesorima. Dakle, transpjuter može da ostvari vezu sa 16 procesora.
  - Između transpjutera nema fizičkih veza, jer smo prije početka rada nekog programa u obavezi da definišemo te veze i to kroz konfiguracione fajlove. Kada startujemo program, mi više nijesmo u mogućnosti da definišemo veze, raskidamo ih i slično. Pogodnost je što procesori nijesu unaprijed povezani.

- **Vrijeme izvršavanja** kod paralelnih računarskih sistema je vrijeme koje protekne od startovanja prvog CPU-a u sistemu do trenutka kad svi procesori završe sa radom.

- Kod paralelnih računara u radu programa prebrojavamo dvije stvari: **korake izračunavanja**, odnosno **računske korake** i **korake usmjeravanja** ili **korake komunikacije** između CPU-a.

- Prilikom izračunavanja vremena izvršavanja radimo aproksimacije i ocjenjivanja da bi smo došli do nekog validnog rešenja. Na primjer, ako su procesori direktno povezani imamo jedan korak, ako imaju jedan međuprocesor, to su dva koraka i tako dalje. Bitno je naglasiti da su koraci komunikacije sporiji od koraka izvršavanja, jer se koraci izvršavanja dešavaju na lokalnom nivou a koraci komunikacije na globalnom nivou jednog računarskog sistema.

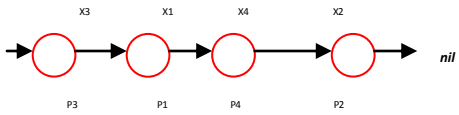
- **Vremenska složenost nekog algoritma** je ukupan broj koraka izračunavanja i koraka usmjeravanja, a preciznija analiza je kada na korake dodamo i njihovu težinu. Kod vremenske složenosti algoritama koristićemo sledeće oznake: za **gornju ocjenu**, za **donju ocjenu** i za **tačnu ocjenu**.

- **Ubrzanje** ili **Speedup,** u oznaci ***s***, je **količnik vremenske složenosti najbržeg poznatog sekvencijalnog algoritma i vremenske složenosti paralelnog algoritma** ($1 \leq s \leq p$, *p – broj procesora u sistemu*).
  s=(V.S.N.P.S.A)/(V.S.P.A)

- **Rad** ili **cijena** ili **Cost,** u oznaci ***c,*** je **proizvod vremenske složenosti paralelnog algoritma i broja procesora u sistemu**.
  c=(V.S.P.A)*p

- **Efikasnost** ili **produktivnost**, u oznaci ***e,*** definišemo kao **količnik vremenske složenosti najbržeg poznatog sekvencijalnog algoritma i cijene algoritma** ili kao **količnik ubrzanja i broja procesora u sistemu**. Važi: , pa slijedi da je . Idealno bi bilo da je $s = p$, pa bi tako bilo $e = 1$, što u praksi ne srećemo.

- **Određivanje ranga (rednog broja) elementa kod ulančane liste koristeći PRAM model sa zajedničkom memorijom sa EREW svojstvom.**
  - Rang (redni broj) elementa liste je broj elemenata koji mu slijede u listi.
  - I-tom elementu liste dodjeljujemo i-ti procesor (u sistemu je n procesora, a u listi n elemenata)

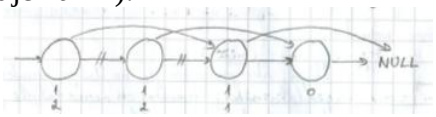


  - Sekvencijalno rešenje ovog problema je složenosti O(n).

$$d[i] = \begin{cases} 0, next[i]=nil \\ d[next[i]+1, inače \end{cases}$$

  - Bolje rešenje je **algoritam sa udvostručavanjem koraka**. Njegova složenost je log(n).
    - Krećemo od poslednjeg elementa kome pridružujemo vrijednost 0 pošto ukazuje na NULL, a preostalim elementima pridružujemo vrijednost 1. Potom svaki CPU uzima vrijednost sebi odgovarajućeg elementa (i-ti procesor uzima vrijednost i-tog elementa).
    - Svaki CPU uvećava svoju vrijednost za vrijednost svog sledbenika i istovremeno raskida vezu sa njim, a pokazivač preusmjerava na sledbenika svog sledbenika.
    - Ako pokazivač elementa u jednom trenutku ukaže na *nil* onda je on dostigao svoju tačnu vrijednost, tj. izračunao je svoj redni broj u listi (rang). Svi elementi koji ne zadovoljavaju pomenuto, vraću se na drugi

korak i ponavljaju ga dok ne zadovolje pomenuto svojstvo (pokazivač mora da im ukazuje na *nil*).



- Formalni zapis prethodnog algoritma (**LIST_RB()**):

```
1. for svaki CPU i (paralelno)
2. do if next[i] = nil
3.    then d[i] ← 0
4.  else d[i]←1
5. while (postoji CPU  i next[i] ≠ nil)
6. do for svaki CPU i (paralelno)
7.   do if next[i] ≠ nil
8.  then d[i]←d[i] + d[next[i]]
9. next[i]←next[next[i]]
```

  - Prva četiri koraka rade lako, bez sinhronizacije i komunikacije i sa složenošću O(1).
  - Od petog do sedmog koraka nemamo ni komunikaciju ni sinhronizaciju, jedino pomenutu sinhronizaciju mreže.
  - U osmom koraku smatramo da imamo neki pojam sinhronizacije i razmatramo ga u dva koraka:
    - $t \leftarrow d[next[i]]$
    - $d[i] \leftarrow d[i]+t$
  - U devetom koraku imamo dva potkoraka:
    - $n \leftarrow next[next[i]]$
    - $next[i] \leftarrow n$
  - Ne može se desiti da dva CPU-a pristupaju istoj lokaciji jer:
    - *$i \neq j$ => $next[i] \neq next[j]$ ili $next[i] = next[j] = nil$ (ovo važi od starta).*

- Dokaz korektnosti algoritma LIST_RB()
  - Za petlju važi sledeća invarijanta:
    Next[i]=nil i d[i]-rastojanje do kraja, tj. d[i] ima svoju vrijednost, ili je next[i]≠nil i next[i] ukazuje na element liste koji je na rastojanju d[i] od njega.
  - Poslije s-tog prolaza kroz petlju ili *d[i]* ima svoju vrijednost i *next [i]=nil* ili je $next[i] \neq nil$ i $d[i] \geq 2^s$.
  - Dokaz invarijante algoritma
    - U s-om koraku imamo: $d[1] \geq 2^s$, $d[2] \geq 2^s$, …
    - U s+1-om koraku imamo:
    - $d[1] \leftarrow d[1] + d[2] \geq 2^s + 2^s = 2^{s+1}$
    - Treba da izaberemo takvo s, tako da je $2^s \geq n$ što nam garantuje da svi procesori čuvaju svoje rastojanje i da je $s = O(\log n)$.

- Složenost, cijena, efikasnost algoritma
  - **Složenost paralelnog rešenja:** *Tp = O (log n)*
  - **Složenost sekvencijalnog rešenja:** *Ts = O (n)*
  - **Ubrzanje:** *S (n) = O (n / log n)*
  - **Cijena:** *C (n) = O (n*log n)*
  - **Efikasnost:** *E (n) = O (*1 */ log n),* 1 */ log n* → *0, kada n* → ∞
  - Ako uzmemo *p = n / log n* procesora i svakom procesoru dodijelimo *log n* čvorova, tj. jedan korak pretvaramo u *log n* koraka, opet imamo složenost *O(log n)*, ali se efikasnost **povećava**:
    *E (n) = n / ((n / log n) * log n)* = 1
    Dakle, **smanjenjem broja CPU-a povećavamo efikasnost**.

- **Zadatak sa prebacivanjem kuglica**
  - Tri kutije A, B i C sadrže redom a, b, c kuglica pri čemu je a*b*c > 0 i b+3c ≥ 12 (konkretno A = 78, B = 109, C = 201). Dozvoljeno je prebacivati kuglice iz jedne u drugu kutiju po sledećim pravilima:
    1. Iz A prebacujemo jednu, a iz C dvije kuglice u kutiju B
    2. Iz B prebacujemo dvije u A, a iz B prebacujemo jednu kuglicu u C
    3. Iz C prebacujemo 4 kuglice u A
    4. Iz A izbacujemo proizvoljan broj kuglica (ne stavljamo ih ni u jednu od ponuđenih kutija)

    Da li je moguće primjenjujući pravila 1-4 isprazniti kutije A, B i C?
  - U opštem slučaju, zadatak možemo riješiti ako i samo ako je broj kuglica u kutiji B djeljiv sa 3 i (B ≥ 9 ili (B ≥ 6 i C ≥ 2) ili (B ≥ 3 i C ≥ 3))
  - U konkretnom slučaju, zadatak nije rešiv jer B nije djeljivo sa 3 (invarijanta zadatka: koje god pravilo da koristimo, B : 3 daje ostatak 1)

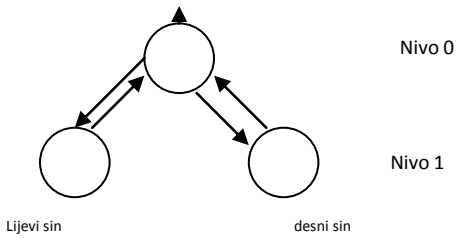
- **Zadatak – računanje prefiksa**
  - Neka je * asocijativna binarna operacija. Na osnovu niza $x_1, x_2, \ldots, x_n$ izračunati niz $y_1, y_2, \ldots, y_n$ po formulama $y_1=x_1$, $y_k=y_{k-1}*x_k=x_1*x_2*\ldots*x_k$, $k=2, 3, 4, 5, \ldots, n$
  - y_i je niz prefiksa.
  - Elementi su smješteni u zajedničku memoriju i povezani su u listu tako da *next[i]* ukazuje na sledeći element. Imamo n procesora i svakom je dodijeljen po jedan element tako da i-ti procesor sadrži i-ti element označen sa *x[i]= x_k*. Pitanje glasi: **kako napraviti paralelni algoritam**?
  - Ako svakom x_i dodijelimo vrijednost 1 i operaciju * zamijenimo operacijom sabiranja, računanje prefiksa se svodi na problem određivanja rednog broja elementa u listi. Uvedimo oznake:

$$[\boldsymbol{i},\boldsymbol{j}] = \boldsymbol{x_i} * \ldots * \boldsymbol{x_j}$$
$$[\boldsymbol{k},\boldsymbol{k}] = \boldsymbol{x_k}$$
$$[\boldsymbol{i},\boldsymbol{k}] = [\boldsymbol{i},\boldsymbol{j}] * [\boldsymbol{j}+\boldsymbol{1},\boldsymbol{k}],\ \ \boldsymbol{1} \le \boldsymbol{i} \le \boldsymbol{j} < k \le n$$
$$\boldsymbol{y}[\boldsymbol{i}] = \boldsymbol{y_k} = [\boldsymbol{1},\boldsymbol{k}]$$

    - $\mathbf{p_i}$ sadrži $x[i] = x_k$ kada važi poslednje

- **Algoritam LIST_PREFIX(X)**
    - for svaki procesor i
    - do y[i] ← x[i]
    - while (postoji i) next[i] != nil
        - do for (svaki procesor i)
            - do if next[i] != nil
                - then y[next[i]] ← y[i]*y[next[i]]
                - next[i] ←next[next[i]]
    - Algoritam možemo realizovati na EREW mašini
    - Složenost algoritma je O(log n) i slijedi iz invarijante
    - Posle s-tog ponavljanja ciklusa pokazivači obuhvataju 2^s čvorova liste ili su *nil*, a k-ti element liste pamti vrijednost

$$[\max(1, k - 2^s + 1), k], \qquad k = 1, \dots, n$$

- 

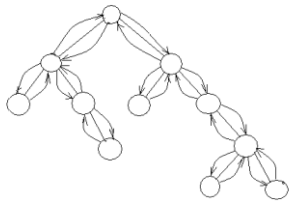
(parent [i], left [i], right[i])
(p[i], l[i], r[i])

- Određivanje dubine (nivoa) svakog čvora u stablu pomoću jednog računara zahtijeva O(n) vremena.

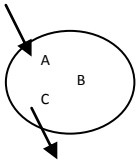
- **Ojlerov put** u grafu je put koji svakom granom grafa prolazi tačno jednom, dok kroz jedan čvor može proći i više puta. Ojlerov put koji počinje i završava u istom čvoru naziva se **Ojlerov ciklus**. Ako imamo 2 čvora neparnog stepena onda Ojlerov put počinje u jednom čvoru, a završava u drugom. Za neorijentisan graf bitni su čvorovi parnog stepena.

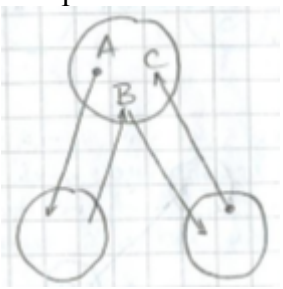
- **U orijentisanom grafu postoji Ojlerov ciklus ako i samo ako se ulazni i izlazni stepen svakog čvora poklapa (graf mora biti i povezan).**

- **Konstrukcija Ojlerovog ciklusa**

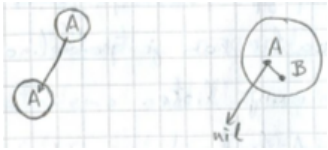


- Svakom čvoru dodjeljujemo 3 procesora A, B, C. Kada dolazimo u čvor uvijek dolazimo u A, a napuštamo ga iz C. B je prolazni put od A do C.

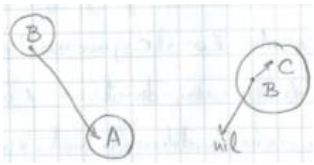




- Ako imamo n čvorova tada imamo 3n procesora; i-tom čvoru odgovaraju 3i, 3i+1, 3i+2 procesori.
- **I:** procesor A ukazuje na procesor A svog lijevog sina ako on postoji, inače ukazuje na procesor B istog čvora.



- **II:** procesor B ukazuje na procesor A svog desnog sina (ako on postoji), inače ukazuje na procesor C istog čvora



- **III:** Procesor C ukazuje na procesor B svoga roditelja, ako je on lijevi sin, a na procesor C svoga roditelja ako je on desni sin. Procesor C iz korijena ukazuje na *nil*

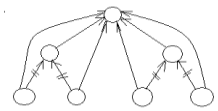
- **Za koliko vremena možemo formirati Ojlerov ciklus**?
  Možemo svaki procesor paralelno povezati sa svojim sledbenikom, a za to nam je potrebno **O(1) vremena** za formiranje liste. Kod ulaza, procesor A treba da ima vrijednost +1, a procesor C vrijednost -1 kod izlaza.
  Ako procesoru A dodijelimo +1, procesoru B 0 i procesoru C -1 i primijenimo algoritam za računanje prefiksa kod kog je * = +, dobićemo da će za svaki čvor procesor C sadržati dubinu čvora, a procesori A i B vrijednost za 1 veću od samog nivoa. Ovo možemo **dokazati matematičkom indukcijom**.
  C u lijevom sinu ima vrijednost kao i B u roditelju, samo što su na razlicitim nivoima.

Složenost algoritma: O(log(3n)) = O(log(n)). Ovo radimo na EREW računaru, pa je **efikasnost: e = n/(3n*log(n)) = 1 / 3log(n)** => slaba efikasnost, veliki broj procesora.

- **Poređenje CRCW algoritma sa EREW algoritmom**
  - CRCW brže i lakše realizujemo od EREW (bolji za praktičnu realizaciju). Između procesora i memorije moraju biti logička kola koja mogu biti veoma složena. U praksi skoro ne postoji ovakav model, postoje njegove aproksimacije.
  - **Konkurentno čitanje – CR**
    - Ilustrovaćemo algoritam konkurentnog čitanja – CR. Primjer ovakvog algoritma je nalaženje korijena svakom čvoru u šumi (šuma je više stabala).
    - Stablo zadajemo tako da svaki čvor ima pokazivač na svog oca.
    - Algoritam za konkurentno čitanje **CRW Find_Roots(F)** (naći korijen u šumi F)

      1. For (za svaki procesor i)
      2. do if p[i] = nil
      3. then root[i] ← i
      4. while (postoji procesor i) p[i] != nil
      5. do for (za svaki procesor i)
      6. do if p[i] != nil (ako je nil vec smo ga odredili)
      7. then root[i] ← root[p[i]]
      8. p[i] ← p[p[i]]

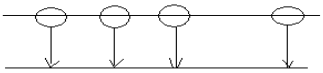
    - Ovdje nema ekskluzivnog čitanja zato što će se desiti da 2 procesora čitaju istu vrijednost.



    - **Složenost algoritma** je O(log(d)) gdje je d visina našeg stabla
    - Koristimo CREW mašinu
    - Koraci 7 i 8 u algoritmu zahtijevaju konkurentno čitanje. Ako imamo n čvorova onda je d ≥ O(log(n)). **Složenost** bi se ponašala kao O(log(log(n))). Ako bi ovo realizovali na bilo kom računaru koji ne zadovoljava konkurentno čitanje, složenost našeg algoritma je Ω(log(n)) (donja ocjena) => manja složenost od log(n)
    - Na svakom nivou mi udvostručavamo broj čvorova.
    - Invarijanta je sledeća:
      Posle s-tog prolaza kroz ciklus ili je p[i] = nil i root[i] je pokazivač korijena ili je p[i] predat na rastojanju 2^s

- **Konkurentno pisanje (CW) – nalaženje maksimuma u nizu**
  - Imamo niz A[1, ..., n] u zajedničkoj memoriji. Imamo **n^2 procesora**. Treba za O(1) vremena (u jednom koraku) naći maksimum ovog niza na mašini CRCW; ovaj model upisuje istu vrijednost
  - Svaki procesor $P_{i,j}$ može da uporedi A[i] i A[j]. Ako je A[i] < A[j], onda A[i] nije max. Potom imamo još jedan niz M[1, ..., n] i on sadrži $\top$ ili $\bot$. Ako je M[i] = true, onda je A[i] = maksimalni element.
  - Možemo napisati sledeći algoritam:
    Fast_max(A)
    1. n ← length(A) (zadavanje dužine niza A)
    2. for i ← 0 to n-1 (paralelno)
    3. do M[i] ← true (u početku su svi kandidati za max element)
    4. for i ← 0 to n-1 and j ← 0 to n-1 (paralelno)
    5. do if A[i] < A[j]
    6. then M[i] = false (korak za istovremeni upis)
    7. for i ← 0 to n-1 (paralelno)
    8. do if M[i] = true
    9. then max=A[i]
    10. return max

    U petom koraku se javlja konkurentno čitanje, tj. $P_{i,j}$ i $P_{j,i}$ traže iste vrijednosti.
    Konkurentni upis se javlja ako npr. procesori $P_{3,7}$ i $P_{3,9}$ pokušavaju da u M[2] stave vrijednost false.
  - Ako primjenjujemo **sekvencijalni algoritam**, da bi našli max niza moramo uporediti sve elemente, tako da je **složenost** O(n)
  - **Složenost Fast_max algoritma**: T(n) = O(1)
    Cijena: $c(n) = O(n^2)$
    Efikasnost: $e(n) = n/n^2 = 1/n \rightarrow 0$ (efikasnost je vrlo slaba)
  - **Da li se efikasnost može poboljšati?**
    Imamo n^2 procesora. $P_{i,j}$ i $P_{j,i}$ $(1 \leq i < j \leq n)$ upoređuju iste članove. Uz malu modifikaciju to sve može da radi isti procesor.
    Stvarno nam je potrebno $\binom{n}{2}$ procesora
    Treba modifikovati algoritam tako da se prvo uporede A[i] i A[j], A[i] != A[j], i vidi koji je manji:
    $A[i] < A[j] \rightarrow m[i] = \bot$
    $A[i] \geq A[j] \rightarrow m[j] = \bot$
    $\binom{n}{2} = (n*(n-1))/2 = O(n^2)$
  - Ako imamo **$n^{3/2}$ procesora naći max niza** za O(1) vremena na CRCW mašini.
    1) Čitav niz se podijeli na više manjih grupa i nađe se max niza
    2) Nalaženje max podgrupa u nizu



- Podijelimo elemente niza u $\sqrt{n}$ grupa sa po $\sqrt{n}$ elemenata. Svakoj grupi pridružimo po n procesora tj. $(\sqrt{n})^2$. Koristeći prethodno opisani algoritam, treba naći max svake grupe za O(1) vremena. Takvih maksimuma ima $\sqrt{n}$ odnosno formirali smo niz dužine $\sqrt{n}$.
- Iskoristimo n procesora da nađemo max ovog niza za O(1) vremena. Dobijeni max je max polaznog niza.
- Pošto imamo $\sqrt{n}$ grupa, i svakoj grupi pridružujemo n procesora, imamo $n^{3/2}$ procesora ukupno. U drugom koraku biće neiskorišćena $n^{3/2}$-n procesora.

  Sada je **složenost**: T(n) = O(1), a **efikasnost**: $e(n) = n/n^{3/2} = 1/\sqrt{n}$

- **Modeliranje CRCW računara pomoću EREW računara**
  - Mašina CRCW je moćnija od EREW mašine.
  - **<u>TEOREMA:</u> za svaki CRCW algoritam (u modelu sa upisom iste vrijednosti) koji koristi p procesora, postoji EREW algoritam za isti zadatak sa istim brojem procesora za koga vrijeme izvršavanja nije veće od O(log(p)) pomnoženo sa vremenom izvršavanja polaznog algoritma**.

    t - vrijeme izvršavanja polaznog algoritma.
    t * log (p) - vrijeme izvršavanja na EREW mašini.
    $t \leq t * \log(p)$

    l,x; l - adresa (prvo polje), x - vrijednost (drugo polje)
    1. Ako procesor p_i želi da upiše vrijednost x_i na lokaciji l_i, onda on u niz na poziciju A[i] upisuje par (l_i , x_i).
    2. Sortiramo niz A po prvoj koordinati l_i .
    3. Procesor p_i upoređuje adresu l_i' koju on ima sa adresom l_{i-1}' koju sadrži procesor p_{i-1}. Ako je $l_i' = l_{i-1}'$ onda procesor p_i ništa ne radi, ako je $l_i' <> l_{i-1}'$ onda procesor p_i upisuje x_i' na lokaciji l_i'.
       Specijalno, procesor p_1 ništa ne radi pošto nema potrebe za upoređivanjem, on će na l_1' upisati x_1'. Na primjer:

2	4
2	4
2	4
5	9
5	9
7	11
7	11
7	11

       Na istim lokacijama upisuje se ista vrijednost.

    Ovo je niz dužine p i možemo ga paralelno sortirati za O(log p) vremena na EREW mašini.

- **Bojenje grafa i max nezavisni skup**
  - Ako 2 procesora pristupaju istom objektu (memorijskoj lokaciji) da bi čitali, a CR nije dozvoljeno, pitanje je kom procesoru dozvoliti da prvi pročita ovaj element
  - Jedan od načina je vjerovatnosni: Svaki od procesora "baca novčić" i u zavisnosti šta "padne" on pristupa ili ne pristupa. (U slučaju konflikta opet se baca novčić sve dok jedan ne dobije prednost). Možemo dati deterministički algoritam za pristup elementu.
  - Posmatra se indeks procesora i onaj sa manjim indeksom ima prednost, više procesora se bore za isti resurs. Ilustrovaćemo pr.: Bojićemo listu sa 6 boja i potom pravimo nezavisan skup.
  - **Bojenje neorijentisanog grafa** G=(V,E) je preslikavanje C: V→N za koje važi (za svako (n, v) ∈ E)C(n) ≠ C(v). Dakle, graf je pravilno obojen ako su svaka dva susjedna čvora u njemu obojeni različitom bojom.
  - Ako je **kardinalnost | C(v)| = k** onda kažemo da smo **graf obojili sa k boja**. Ponekad je pogodno slikati u skupu $N_0 = N \cup \{0\}$
  - **PARALELNI ALGORITAM ZA BOJENJE GRAFA U OBLIKU LISTE**

    Interesuje nas bojenje sa što je moguće manje boja (listu bojimo sa najmanje 2 boje)

    Primijetimo da se ova lista oboji sa 6 boja, skoro za konstantno vrijeme.

    Konstruišemo za bojenje liste niz bojenja Co, ..., Cn, pri cemu Ck+1 konstruišemo pomoću Ck tako da Ck+1 koristi manje boja od Ck.

    Pretpostavimo da imamo n procesora i n elemenata i da svaki procesor zna svoj index.

    Ako neki procesor sadrži element x, onda njegov index označavamo sa $P(x)*C_0$ koristeći n boja i to su indexi procesora, tj. $C_0(x) = P(x)$ (ovo je tkz. lažno bojenje). Opisati indukcijski korak, ako smo već izračunali Ck i treba izračunati Ck+1.

    Pretpostavimo da u bojenju Ck boje mozemo zapisati sa r bita.

    Neka je Ck(x) = *a*, a Ck(next(x)) = *b*, pri čemu je $a = (a_{r-1}a_{r-2}....a_1a_0)_2$, $b=(b_{r-1}b_{r-2}...b_1b_0)_2$ zapisano u binarnom brojnom sistemu. *a* i *b* su boje i ovo je njihova binarna reprezentacija. Posto su x i next(x) različito obojeni, boje *a* i *b* su različite, jer su boje susjednih elemenata => da (postoji i ∈ {0,...,r-1} takvo da je $a_i \neq b_i$.

    Novu boju za x, odnosno $C_{k+1}(x)$, definisaćemo kao par $(i, a_i)_2$, $C_{k+1}(x) = (i, a_i)_2 = i*2 + a_i$ (ako ima više indeksa, možemo uzeti prvi ili bilo koji od njih), tj. gledamo binarnu reprezentaciju zapisa i dopisemo cifru a_i.

    Bojenje za poslednji element u listi (nema sledbenika) je $(0, a_0)_2$, tj. to je boja dobijena od a_0, odnosno od bojenja C_k dobili smo C_{k+1} (biće ili 1 ili 0). (Ne mijenja se vremenski, svaki put je ista cifra).

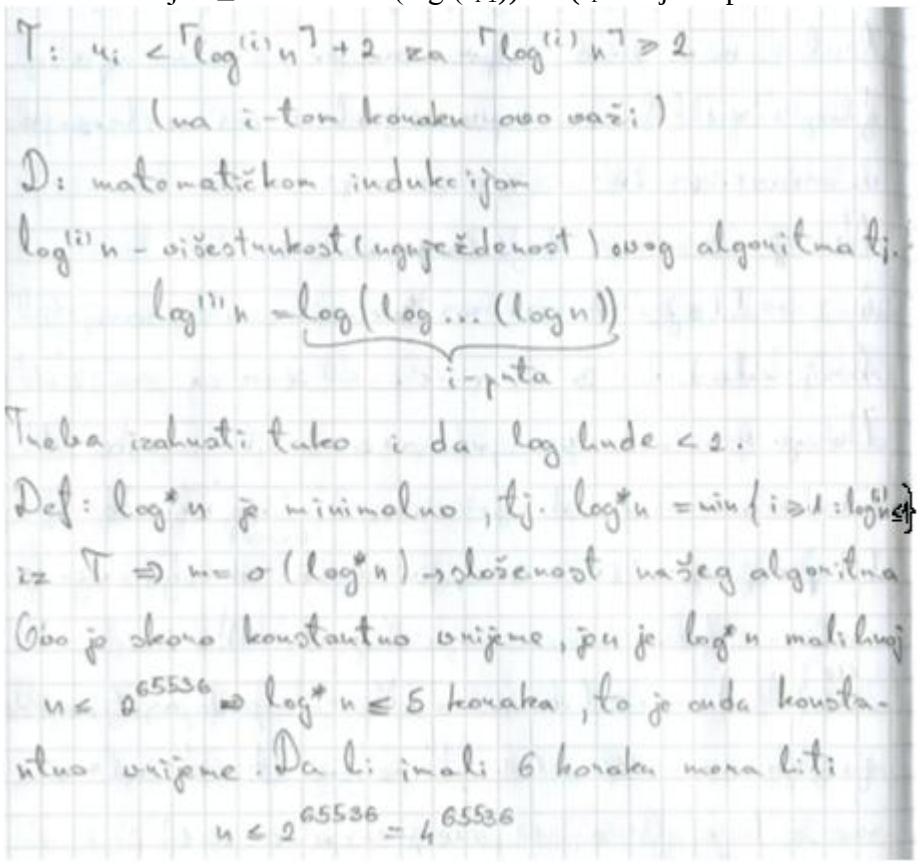
    Sa koliko bita možemo zapisati novu boju?

    Pošto je i iz {0, 1,..., r-1}, za njegovo binarno zapisivanje nama treba log(r) bita, a pošto dopisujemo cifru a_0 treba nam još bit, pa nam je potrebno najviše log(r + 1) bita za zapisivanje boja u Ck+1.

    Ako je $r \geq 4 \rightarrow$ nad cio dio (log(r)) +1 = 2+1=3

    $r = 3 \rightarrow$ nad cio dio (log3) +1 = 2 + 1 = 3

Ako je r = 3 sve je zapisano sa 3 bita $a_2a_1a_0$. Naši kodovi mogu da počinju sa i iz 00/0 01/1 10/2 (dopisujemo 0 ili 1 u zavisnosti od toga koja je naša cifra i imaćemo 6 boja).
Kad su 3 bita otpadaju boje 110 i 111. Naše boje su u opsegu od 0 do 5: (000, 001, 010, 011, 100, 101).
Koliko nam treba koraka do finalne boje?
Co, .., Cm m = ?
Formiranje jedne boje ide od O(1) => za ova bojenja nam je potrebno O(m) koraka, odnosno koliko je različitih bojenja. Ako u i-tom koraku koristimo r_i bita, onda važi da je i ≤ nad cio dio (log (r_{i-1})) +1 (r_i manji od prethodnih koraka).

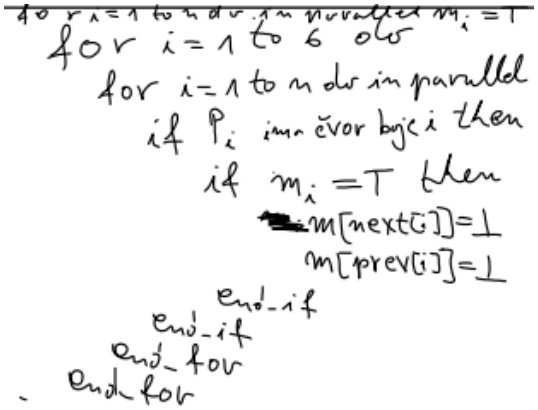


- **NEZAVISAN SKUP**

  Skup V'c V čvorova grafa G= (V,E) nazivamo **nezavisnim skupom**, ako za svako (n,v) iz V', (n,v) ne pripada E, tačnije nikoja 2 vrha iz V' nijesu spojena granama iz skupa E.
  Nezavisni skup V' nazivamo **maksimalnim nezavisnim skupom** (**MNS**) ako dodavanjem bilo kog vrha iz V/V' on prestaje da bude nezavisan skup (ne treba ovo miješati sa **najvećim nezavisnim skupom** koji predstavlja nezavisni skup najveće moguće veličine za dati graf).
  Nalaženje najvećeg nezavisnog skupa je NP-težak problem.
  Nas interesuje kako naći maksimalni nezavisni skup.

Ako gledamo 3 elementa (susjedna), onda bar jedan element pripada maksimalnom nezavisnom skupu, tj. svaki maksimalni nezavisan skup sadrži više od n/3 elemenata. Prvo obojimo listu sa 6 boja, a onda svaki element liste označimo sa tačno, tj n[i]=T. Za svaku od 6 boja ponovimo sledeće korake (to radimo paralelno):

Ako razmatramo boju k, onda gledamo da li je C(x) = k i da li je n(x)=T. Ako on ispunjava ova 2 uslova, onda mi x dodajemo u MNS, a njegovog prethodnika i sledbenika označimo tako da ih ne mozemo ukljuciti u MNS – (n(next(x))=(neT) i n(prev(x))=(neT), i proces ponovimo za svih 6 boja. Koraci se izvrsavaju paralelno.

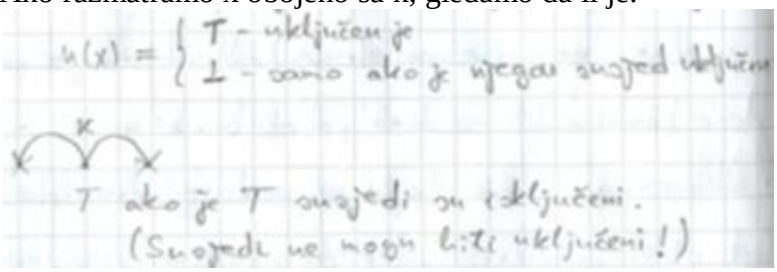


Za ovu proceduru ukupno nam treba o(1) vremena.
Složenost algoritma je o(log n) – za početni korak, da bi listu obojili sa 6 boja.
Dokaz da je algoritam korektan, tj. da izdvaja maksimalni nezavisni skup:

Svako x je obojeno nekom bojom i bar jedan od 6 koraka će doći na razmatranje.

Ako važi T – uključen je, a ako važi neT nije jer je njegov susjed uključen.

Ako razmatramo x obojeno sa k, gledamo da li je:

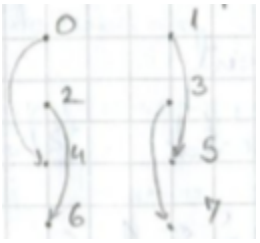


- **Data broadcasting**
  - **Data broadcasting** (predaja podataka) se vrši kada jedan procesor želi da preda podatke ostalim procesorima.
  - Postoje dva tipa predaje podataka:
    1) Jedan – svima: kada jedan procesor želi da proslijedi svoje podatke svim ostalim procesorima.
    2) Svaki – svakom: Svaki od p procesora šalje svoj podatak svim ostalim procesorima

Ovo možemo da realizujemo na CREW i CRCW mašinama.

- Složenost po tipovima je:
  1) o(1)
  2) o(p) vremena za CREW.

  Ako sve realizujemo na EREW mašini, 1. slučaj realizujemo za o(p) vremena, ali može i brže.



Za svega 3 koraka, početna informacija je proslijeđena do 8 procesora.

- U memoriji rezervišemo niz B[j]. Tu će biti smještena kopija podataka koja je namijenjena procesoru p*j*. Algoritam prvo svoju informaciju smješta na lokaciju B[1], pa B[2], pa onda paralelno u sledećem koraku B[3] i B[4].
- **Algoritam BROADCAST (D,p,B):**

  B[1] ← D
  For k=0 to (nad cio dio ($\log_2(p)$) - 1 do
    For j = 2^k+1 to 2^{k+1} do in parallel
      B [j+2^k] ← B[j]
    End_for
  End_for
- Složenost ovog algoritma je O(log(p)), a unutrašnja petlja zahtijeva o(1) vremena. Da li se ovo može brže uraditi? Ne može, jer nije dozvoljeno konkurentno čitanje, a mi od k podataka stvaramo 2k. Dakle, složenost ovog problema je O(log(p)).

- Kako da realizujemo na EREW mašini tip svaki-svakom?

  Ako dozvolimo CRCW, ovaj tip realizujemo za O(log(p)) vremena. Imamo jedan niz B[j], j=1…p procesora. Prvo procesor 1 proslijedi svoju vrijednost procesoru 2, itd, odnosno svakom sljedbeniku se prosleđuje svoja vrijednost: 1 → 2, 2→ 3… (ciklično prosleđivanje podataka).

- **Algoritam BROADCAST_ALL(p, B):**

  For (za svaki) procesor j do
    P_j upisuje svoj podatak u B[j]
  End_for

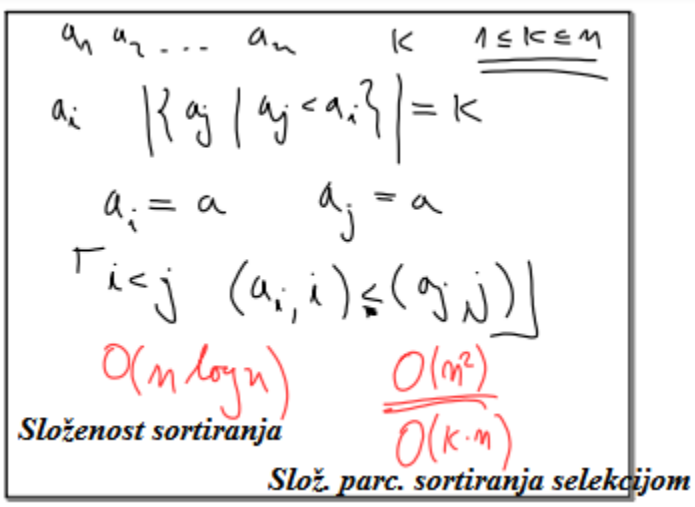
  For k=1 to p-1 do
    For (za svaki) procesor j do
      P_j čita podatak B[(j-k) mod p]
    End_for
  End_for
- Složenost prethodnog algoritma je O(p), dok svaka unutrašnja petlja zahtijeva o(1) vremena za realizaciju.

- **Algoritam sortiranja za širenje podataka (EREW algoritam)**
  - Neka imamo niz sa elementima $S_{1...}S_n$. Sortiranje se vrši tako što prvo sračunamo poredak (za svaki) element, tj. poziciju elementa gdje se nalazi tako što gledamo koliko elemenata je manje od njega.
  - **Algoritam:**

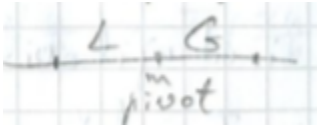
```
For (za svaki) procesor j do R[j] ← 0

For k=1 to n-1 do
        For (za svaki) procesor j do
                l ← (j+k) mod p
                If (S[l] < S[j]) or (S[l] = S[j] and l < j)
                        Then R[j] ← R[j]+1
                End_if
        End_for
End_for

For (za svaki) procesor j do S[R[j]] ← S[j]
```

    - Složenost unutrašnje petlje je o(1). (ukupna složenost – o(n) ???)

- **Algoritam selekcije**
  - Neka imamo niz S od n elemenata. Dalje imamo element S[j] i dat je broj k, $1 \leq k \leq n$. Treba naći k-ti po veličini broj u nizu S, pri čemu uzimamo da, ako se traži prvi po veličini, tražimo min, a ako se traži k-ti, tražimo max. Ako nam treba srednji, kako njega odrediti?



Imamo jedan niz, podijelimo ga na dva manja i biramo element.



Grupišemo elemente manje od pivota i veće od pivota. Ako je pivot na poziciji m i k < m, mi nastavljamo trazenje k-tog elementa u prvoj polovini, tj. u L i time smo eliminisali n-m (ako smo odabrali srednji element – ½) naših elemenata. Ako je k > m, trazimo (k - m) = i-ti element u nizu G. Ako bi pivot uvijek bio na sredini, onda bi **složenost** našeg algoritma zapisali kao:

T(n) = T(n/2) + O(n) => za O(n) vremena određujemo traženi element.

U opštem slučaju je T(n) = T(max{|L|, |D|}) + O(n). Najgori slučaj je: T(n) = T(n-1) + O(n) = $O(n^2)$.

Dakle, prosječna složenost je T(n) = O(n) (pod pretpostavkom da uvijek biramo u nizu S k-ti element po veličini).

- **Algoritam Select (S,K)**

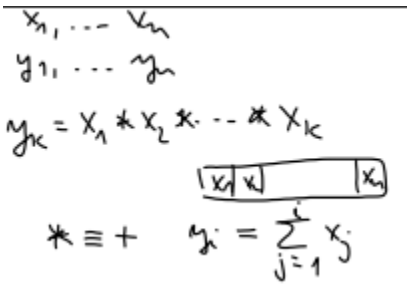
1. if |S| < q (q ranije zadato)
   then sort(S)
   return S[k]
   else
   podijeli S u |S|/q nizova velicine q, sortiraj ih i od njihovih medijana formiraj niz M velicine |S|/q (složenost – O(n))
2. m = select (M, |M|/2) (tražimo medijan)
3. kreiraj 3 podniza:
   L: elemente S koji su < od m
   E: elemente S koji su = m
   G: elemente S koji su > m
4. if |L| >= K then
   return Select (L, K)
   else if |L| + |E| >= K
   then return k
   else
   return Select (G, K-|L|-|E|)
   end_if
   end_if

   - |S| = n, q > 4
     T(|S|) = O(|S|) + T(|S|/q) + O(|S|) + T(max{|L|, |G|})
     |M| = |S|/q, |L| > |S|/4, |G| > |S|/4
     |L| < |S| - |G| < ¾|S|
     |G| < ¾|S|
     T(n) ≤ T(n/q) + T(3n/4) + O(n)
     T(n) ≤ d*n
     T(n) ≤ T(n/q) + T(3n/4) + c*n ≤ d/q * n + 3n/4*d + c*n = n (d/q + 3/4*d + c) ≤ d*n
   - q = 5
     d/5 + 3/4*d + c ≤ d /*20
     19d + 20c ≤ 20 d
     20 c ≤ d
   - T(n) ≤ d*n, pa je T(n) = O(n)

- **Problem podjele skupa tačaka pravom**
  - Imamo pravu p i n tačaka u ravni. Treba povući pravu paralelnu sa pravom p tako da pola talaka bude sa jedne, a pola sa druge strane.

- O(n)
  (x_i,y_i)
  $Ax + By + C = 0$
  (A,B,C)

- **Algoritam ALL_SUMS (A, N)**




```
ALLSUMS(A,N)
for j=0 to logN-1 do
  for i=2^j+1 to N do in parallel
    A(i) ← A(i)+A(i-2^j)
  end_for
end_for
```

- **Paralelni algoritam selekcije**
  - $p = n^{1-x}$, $0 < x < 1$
    Za $x = ½$: $p = \sqrt{n}$
    $O(n^x) = O(n/p)$
  - **Algoritam PRAM Select(S,p,k)**
    Imamo skup S od n elemenata, koristimo $p=n^{1-x}$ procesora pri čemu je $0<x<1$.
    1) if $|S| < 4$
       then sort S
       return S[k]
       else
       Dijelimo S u p podnizova $S^{(j)}$ veličine $|S|/p$
       end_if

       - **Složenost** ovog koraka je O(1)).

    2) Broadcast |S| svim procesorima
       (za svako) j processor p_j paralelno računa $M_j \leftarrow$ Select($S^{(j)}$, nad cio dio ($|S^{(j)}|/2$)), gdje je M niz dužine p → Procesor p_j računa M_j tako što izvrši operaciju Select medijane iz niza $S^{(j)}$.

- Da bi kardinalnost S proslijedili svim procesorima treba nam logp vremena. Prvi procesor uzima |S|/p, drugi od pozicije 2|S|/p itd... Svaki procesor može da proračuna koji dio niza njemu pripada.
  Imamo da je$|S^{(j)}|= |S|/p = n/n^{1-x} = n^x$; Ne pretpostavljamo da je x=0 jer bi u tom slučaju dobili 1 i u nasem koraku ne dominira n^x već log p.
  Ukupno za ovaj korak nam treba $O(n^x)$ vremena i dobijamo $logp = log\ n^{1-x}$; ako je x=1 dobijamo jedan procesor, pa to ne bi bio paralelizam.

3) m ← PRAMselect(M, p, nad cio dio (|M|/2))

   - U ovom koraku radimo i pozivamo algoritam paralelno, tako da složenost zavisi od |M|: $T(n^{1-x})$.

4) Broadcast m svim procesorima – Broadcast (m, p, B)
   Vršimo podjelu dobijenih nizova na podnizove
   L: elementi iz S koji su < od m
   E: elementi iz S koji su = m
   G: elementi iz S koji su > od m

   - Za izvršavanje procedure Broadcast (m, p, B) potrebno nam je O(logp) vremena, odnosno $O(n^{1-x})$ vremena. Za podjelu niza S na podnizove L,E,G treba nam $O(n^x)$ vremena. Svaki od procesora zna vrijednost m.
     $L^{(j)},E^{(j)},G^{(j)}$ → svaki procesor to uradi lokalno; određivanje svakog od podnizova $L^{(j)}$ odvija se paralelno i svaki od procesora može sračunati svoju dužinu $|L^{(j)}|$.

5) If |L| ≥ k
   then return PRAM Select(L,p,k)
   else
   if |L|+|E| ≥ k
   then return m
   else
   return PRAM Select (G,p,k-|L|-|E|)
   end_if
   end_if

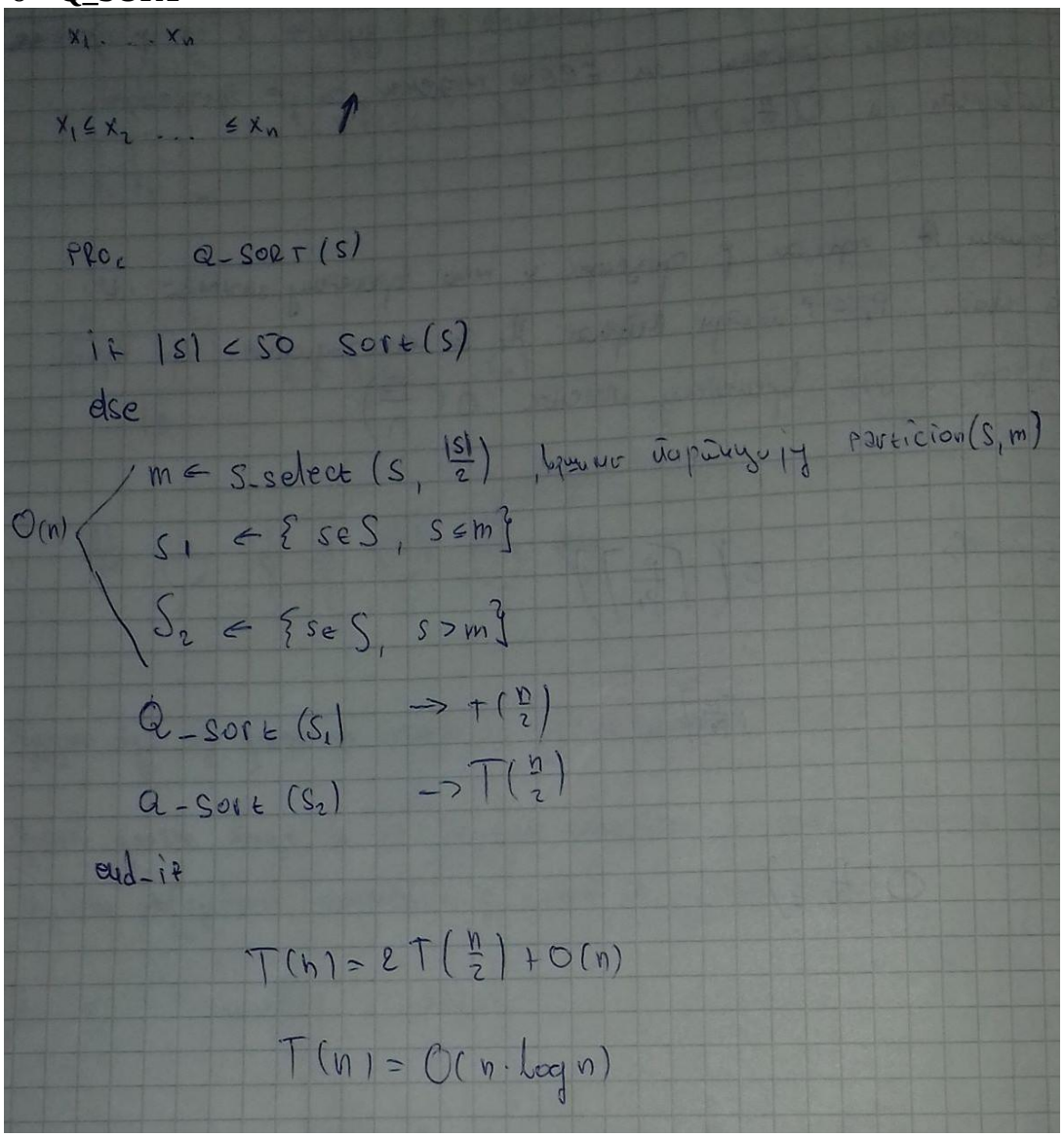
   - Moramo sračunati sa koje pozicije procesor treba da počne sa upisivanjem. Podnizovi nijesu iste kardinalnosti. Procesor zna od koje pozicije da upisuje na osnovu paralelnog algoritma koji radi logaritamski pa nam tu treba O(log p) vremena.
   - Samo za upisivanje treba nam $O(n^x)$ vremena. Svi procesori mogu da upisuju na određene lokacije. Da bi se upisala sva tri podniza, treba nam n^x koraka, jer je zbir tih nizova u stvari $S^{(j)}$.

o **Ukupna složenost** može da se izrazi kao:
  $T(n,p) = T(p,p) + T(3/4n,p) + O(n^x)$
  - Ovo možemo pretvoriti u relaciju od n. Dobija se:

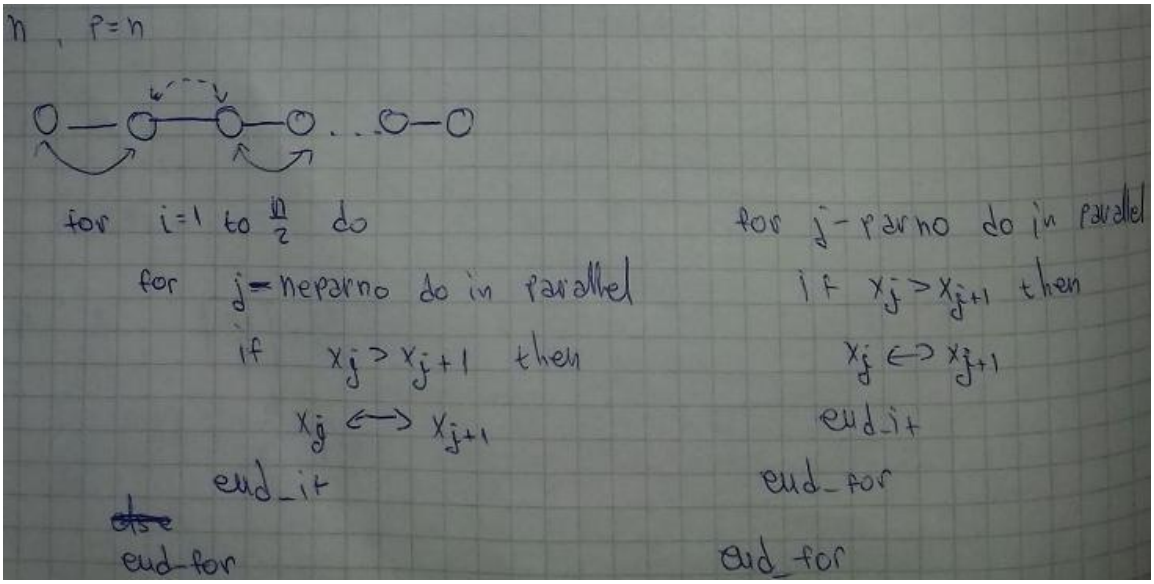
$T(n,p) = O(n^x)$

- Složenost sekvencijalnog algoritma je $Ts(n) = O(n)$.
- Onda je **ubrzanje**: $s(n,p) = n/n^x = n^{1-x} = p$
- **Efikasnost** $E=1 \rightarrow$ dobar algoritam!

- **Sortiranje**
  - **Q_SORT**

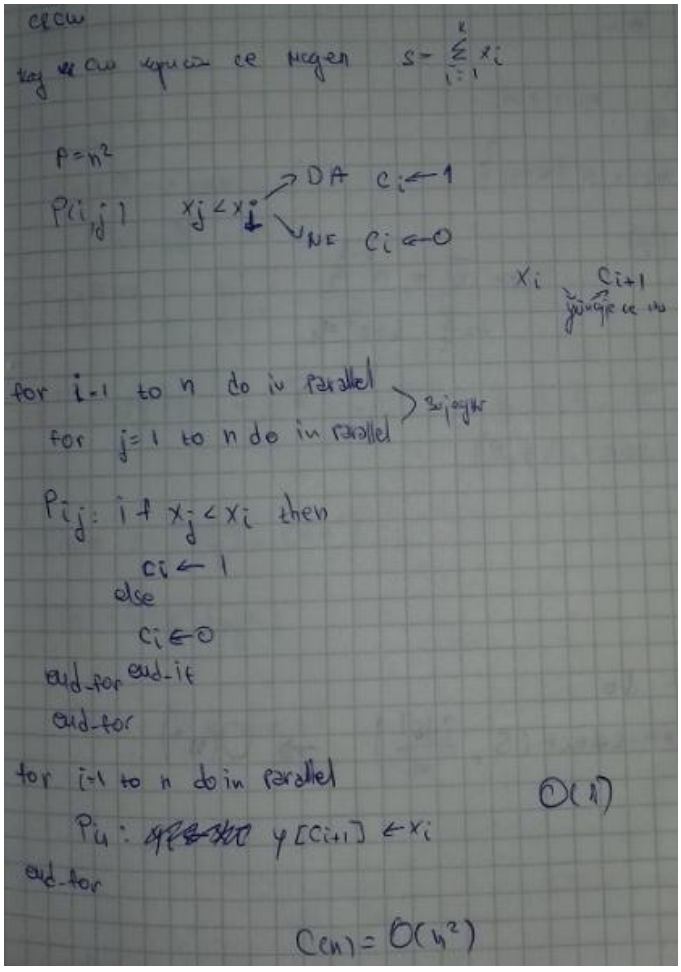


o

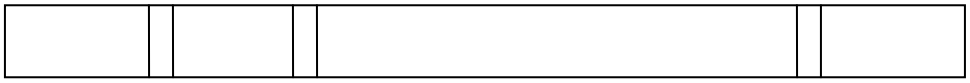


Složenost: O(n/2) = O(n)
Cijena: C(n) = $O(n^2)$

o



- **Paralelni algoritmi sortiranja**
  - Imamo neki niz brojeva $x_1,\ldots,x_n$ i treba da izvršimo takvu permutaciju elemenata da dobijemo $y_1,...,y_n$ takvih da je $y_1 \leq \ldots \leq y_n$. Postoje različiti paralelni algoritmi za sortiranje. Prvo fiksiramo jedan broj k i izaberemo k-1 elemenata niza. Neka su to elementi $m_1,\ldots,m_{k-1}$. Pretpostavićemo da je m_0=-beskonačno, m_k=+beskonačno (posmatramo kao brojanje, m_k je element veći od svih naših elemenata). Ovi elementi će činiti particiju našeg niza. Pretpostavimo da je $m_i < m_{i+1}$.



Ove particije će biti različite veličine (a mi ćemo nametnuti da budu iste), jer brojeve $m_1,\ldots,m_k$ biramo nasumice. Pretpostavimo da je (i*n/k)-ti element niza izabran da bude m_i. Pomoću m_i dobićemo k particija koje će biti iste veličine (svaka ima n/k elemenata).
Kad izvršimo particionisanje pretpostavljamo da su elementi lijevo od m_i manji od m_i, a elementi desno od m_i veći od m_i.
Ostaje nam da svaku od particija $[m_i,m_{i+1})$ sortiramo rekurzivno na isti način i time ćemo dobiti sortirani niz.
Pretpostavka je da imamo $p=n^{1-x}$ procesora i da je $0 < x < 1$. Takođe pretpostavljamo da biramo broj k (unaprijed fiksirani broj) koji ćemo izabrati tako da je $k=2^{1/x}$ (particiju vršimo na k djelova), npr. $x = 1/2$, $p = n^{1/2}$, $k = 2^2 = 4$.

  - **Algoritam Pram_Select_Sort(S,p):**

1. If |S|<k
   then return Q_Sort (S)
   end_if
2. (izbor brojeva m_i)
   For i=1 to k-1 do
   m_i = PRAM_Select(S, i*|S|/k, p)
   {m_o=-beskonačno, m_k=+beskonačno}
   End_for
3. (Vršimo particiju pomoću brojeva m_i)
   For i=0 to k-1 do
   Konstruiši podniz $S^{(i)}$ od elemenata iz S koji su po veličini između brojeva m_i i m_{i+1} (paralelno) (broadcast svih m_i)
   End_for
4. For i = 1 to k/2 do in parallel
   PRAM_Select_Sort($S^{(i)}$, 2p/k procesora)
   (jer istovremeno vršimo sortiranje k/2 podnizova, tj. pošto za svaki podniz koristimo po 2p/k procesora, imamo dovoljno procesora za k/2 podnizova)

End_For

For i = (k/2 +1) to k do in parallel
PRAM_Select_Sort($S^{(i)}$, 2p/k)
End_For

Kardinalnost $S^{(i)}$ je povezana sa brojem procesora $|S^{(i)}|^{1-x} = 2p / k$.
1.korak nas košta konstantno vrijeme – O(1).
U 2. koraku k puta pozivamo paralelni select, pa nas to košta $k*O(n^x)$, pa je $O(n^x)$ je složenost za drugi korak.
U 3. koraku imamo niz u zajedničkoj memoriji i određeni broj procesora koji je manji od veličine niza .
Algoritmom broadcasting ćemo proslijediti svim procesorima veličinu niza S. Na osnovu toga svaki procesor će proračunati koji dio niza njemu pripada. Svaki procesor uzima |S|/p elemenata i nezavisno vrsi particiju svoga dijela.
Jedan procesor posmatra jedan dio niza i na njega vrsi particije koje mogu biti različitih velicina i računa veličinu svake particije. Primijenićemo k puta paralelni prefix algoritam i dobijamo informaciju od koje do koje pozicije procesor smješta svoje elemente u odredjenu particiju.
Da bi S proslijedili svim procesorima treba nam logp, a konstantno vrijeme da odredi svoj dio niza.
Procesor ima $|S|/p=n/n^{1-x}=n^x$ elemenata. Da bi izvrsio particiju n elemenata treba $O(n^x)$ vremena.

$$[\log p < O(n^x), => p < n]$$

Koliko nam vremena treba za paralelni prefix algoritam?
Pošto imamo p procesora, potrebno nam je je O(log p) vremena. Svaki procesor treba da prepiše svoje elemente na određene particije za $O(n^x)$ vremena za sve elemente. Koliko nas kosta 3. korak? Odgovor je $O(n^x)$.
Složenost jednog poziva paralelnog algoritma u četvrtom koraku je:

$$T(n/k, 2p/k)\ |S^{(i)}| = n/k$$

Čitav 4. korak nas košta $2*T(n/k, 2p/k)$.

Ukupna složenost je $T(n,p)= O(n^x) + 2*T(n/k,2p/k)$. Imamo u vidu da je $p=n^{1-x}$.

$$(n/k)^{1-x} = (n/2^{1/x})^{1-x} = n^{1-x}*1/2^{1-x/x}= 2*n^{1-x}/2^{1/x} = 2p/k$$

Tačno dobijamo vezu između n/k i 2p/k.
Ova relacija nam daje:

$$T(n,p)= O(n^x \log n)$$
$$T_s(n)= O(n \log n)$$

$s(n,p) = n \log n/n^x \log n = n^{1-x} = p$ => ubrzanje je maksimalno.
Cijena ili rad $C(n,p)=O(n \log n)$ => algoritam ima dobre performanse.

Pretpostavka je da je broj procesora manji od dužine niza, inače nećemo imati dobru efikasnost.

- **Algoritam PRAM RandomSort(S,p)**
  1. For svaki procesor k do
     Uzima $|S|/p^2$ elemenata od svojih $|S|/p$ elementa i smješta na odgovarajuću poziciju u nizu R
     End_For
  2. Procesor p_0 sortira niz R(određuje m_i)
     $m_i = (i * |S|/p^2)$ – ti element R
  3. For svaki procesor i do
     Smješta svoje elemente između m_i i m_{i+1} u $R^{(i)}$
     End_For
  4. For svaki procesor j do
     Sort($T^{(i)}$): i-ti procesor uzme $R^{(i)}$ i sortira ga.
     End_For

Merge

- Neka su data dva niza A=(a1, a2, a3...) i B = (b1,b2,b3...) i neka su oba niza sortirana u rastućem poretku.
  Treba formirati niz C nadovezivanjem nizova A i B tako da i niz C bude sortiran.
  Formiranje niza C se može obaviti sekvencionalnim algoritmom. Algoritam koristi dva pokazivača *i* i *j* koji su inicijalno postavljeni na prvi element niza A i prvi element niza B.
  Upoređuju se elementi na koje pokazuju pokazivači i manji element se smješta u niz C, zatim se pokazivač koji je sadržao manji element uveća za 1... Ovaj postupak se nastavlja sve dok se jedna ulazna sekvenca ne isprazni odnosno dok ne dođemo do kraja jednog niza. Kada stignemo do kraja jednog od nizova ostatak drugog niza se prepiše u niz C. Neka je r dužina niza A, a s dužina niza B.

```
procedure SEQUENTIAL MERGE (A, B, C)

Step 1:  (1.1) i ← 1
         (1.2) j ← 1.

Step 2: for k = 1 to r + s do
                if a_i < b_j then (i) c_k ← a_i
                                  (ii) i to i + 1
                             else (i) c_k ← b_j
                                  (ii) j ← j + 1

                end if
        end for. □
```

  Složenost ovog algoritma je O(r+s) jer ćemo imati r+s poređenja.
- **Parallel Merging**
  CREW mašina se sastoji od N procesora P1...PN. Treba dizajnirati paralelni algoritam za ovaj kompjuter koji uzima dva niza A i B kao ulaz i kao izlaz daje niz C. Bez gubljenja opštosti možemo predpostaviti da je r <= s.
  Algoritam koristi N procesora gde je N manje ili jednako r i najgori slučaj gdje je r=s=n ima složenost: O((n/N) + log n) vremena.

Svaki od N procesora je sposoban da izvrši dvije procedure: SEQUENTIAL MERGE i BINARY SEARCH.

- **Binarna pretraga** kao ulaz uzima niz S = (s1,s2...sn) brojeva sortiranih u neopadajućem redosledu i broj x. Ako x pripada S procedura vraća indeks k elementa s_k u S tako da je $x=s_k$. Inače procedura vraće nulu. Binarna pretraga je bazirana na principu podijeli pa vladaj. Na svakom nivou vrši se upoređivanje između elementa x i elementa niza S.

procedure BINARY SEARCH (S, x, k)

Step 1: (1.1) $i \leftarrow 1$
(1.2) $h \leftarrow n$
(1.3) $k \leftarrow 0$.

Step 2: **while** $i \leqslant$ h **do**
(2.1) $m \leftarrow \lfloor (i+h)/2 \rfloor$
(2.2) **if** $x = s_m$ **then** (i) $k \leftarrow m$
(ii) $i \leftarrow h+1$
else if $x < s_m$ **then** $h \leftarrow m-1$
else $i \leftarrow m+1$
end if
end if
end while. □

Pošto se broj elemenata upola smanjuje za svaki korak procedura zahteva O(log n) vremena.

- **Procedure CREW MERGE (A,B,C)**

Korak 1: Selektuje se N–1 elemenata niza A koji dijele niz na N podnizova približno iste veličine. Nazovimo podniz formiran od ovih N–1 elemenata A/. Podsekvenca B/ formira se slično.

for i = 1 **to** $N-1$ **do in parallel**
Processor P_i determines a'_i and b'_i from
(1.1) $a'_i \leftarrow a_{i\lfloor r/N \rfloor}$
(1.2) $b'_i \leftarrow b_{i\lfloor s/N \rfloor}$

end for.

Korak 2: Radimo merge sekvenci A/ i B/ u sekvencu trojki
$V = \{v_0, v_1, \ldots, v_{2N-2}\}$, gdje se svaka trojka sastoji od elementa niza A/ ili B/ , njegove pozicije u nizu i imena njegove originalne sekvence.

(2.1) **for** i = 1 **to** $N-1$ **do in parallel**
(i) Processor P_i uses BINARY SEARCH on B' to find the smallest j such that $a'_i < b'_j$

(ii) if j exists then $v_{i+j-1} \leftarrow (a'_i, i, A)$
else $v_{i+N-1} \leftarrow (a'_i, i, A)$
end if
end for
(2.2) for i=1 to N−1 do in parallel
(i) Processor P_i uses BINARY SEARCH on A' to find the smallest j such that $b'_i < a'_j$
(ii) if j exists then $v_{i+j-1} \leftarrow (b'_i, i, B)$
else $v_{i+N-1} \leftarrow (b'_i, i, B)$
end if
end for.

Korak 3: Svaki procesor vrši merge i unosi u niz C elemente iz dvije podsekvence, jedne iz A i jedne iz B. Indeksi od dva elementa (jednog u A, drugog u B) se prvo izračunavaju i unose u niz Q.

(3.1) $Q(1) \leftarrow (1, 1)$
(3.2) for i=2 to N do in parallel
if $v_{2i-2} = (a'_k, k, A)$ then processor P_i
(i) uses BINARY SEARCH on B to find the smallest j such that $b_j > a'_k$
(ii) $Q(i) \leftarrow (k\lceil r/N \rceil, j)$
else processor P_i
(i) uses BINARY SEARCH on A to find the smallest j such that $a_j > b'_k$
(ii) $Q(i) \leftarrow (j, k\lceil s/N \rceil)$
end if
end for
(3.3) for i=1 to N do in parallel
Processor P_i uses SEQUENTIAL MERGE and $Q(i) = (x, y)$ to merge two subsequences one beginning at a_x and the other at b_y and places the result of the merge in array C beginning at position x + y − 1. The merge continues until
(i) an element larger than or equal to the first component of v_{2i} is encountered in each of A and B (when $i \leqslant N-1$)
(ii) no elements are left in either A or B (when i=N)
end for. □

Korak 1 zahtijeva konstantno vrijeme.
Korak 2 se sastoji od dva poziva binarne pretrage sekvence dužine N – 1. Zahtijeva O(logN) vremena.
Korak 3:

Korak 3.1 se izvršava za konstantno vrijeme.
Korak 3.2 zahtijeva vrijeme od O(logN).
Korak 3.3, posmatramo prvo V koje sadrži 2N – 2 elemenata koji dele C na 2N – 1 podsekvenci maksimalne veličine $(\lceil r/N \rceil + \lceil s/N \rceil)$. U koraku 3 svaki procesor kreira dvije takve podsekvence niza C čija ukupna veličina nije veća od $2(\lceil r/N \rceil + \lceil s/N \rceil)$, osim jednog procesora koji

kreira samo jednu podsekvencu niza C. Sekvencionalni merge ima složenost O((r + s)/N).
U najgorem slučaju kada je r=s=n, s obzirom da je n veće ili jednako od N, algoritmom dominira vrijeme u trećem koraku.

$$t(2n) = O((n/N) + \log n).$$

Kako je p(2n)=N, c(2n)=p(2n) x t(2n) = O(n+N log n), algoritam je optimalan kada je $N \leqslant n/\log n$.

- **Nalaženje medijana dva sortirana niza**

Neka su data dva niza A=(a1, a2, a3...) i B=(b1,b2,b3...) gdje je r,s >=1 i neka rezultat spajanja nizova ima dužinu m=r+s (A.B). Traži se medijan koji se nalazi na poziciji m/2 (cio dio nad) niza A.B bez formiranja niza A.B. Algoritam vraća par (a_x,b_y) koji zadovoljava sledeće osobine:

1. Ili je a_x ili b_y medijan niza A.B odnosno ili je a_x ili b_y veće od $\lceil m/2 \rceil - 1$ elemenata i manje od tačno $\lfloor m/2 \rfloor$ elemenata.
2. Ako je a_x medijan onda b_y je ili:
   a) Najveći element u B manji ili jednak od a_x, ili
   b) Najmanji element u B veći ili jednak od a_x.

   U suprotnom ako je b_y medijan tada je a_x ili :
   a) Najveći element u A manji ili jednak od b_y, ili
   b) Najmanji element u A veći ili jednak od b_y.
3. Ako više od jednog para zadovoljavaju prvi i drugi uslov algoritam vraća par gde je x+y najmanje.

   Označimo sa (a_x,b_y) medijan par niza A.B. x i y su indeksi medijan para. a_x je medijan niza A.B ako važi:

   (i) $a_x > b_y$ and $x + y - 1 = \lceil m/2 \rceil - 1$ or
   (ii) $a_x < b_y$ and $m - (x + y - 1) = \lfloor m/2 \rfloor$.

   Inače je b_y medijana niza A.B.

```
procedure TWO-SEQUENCE MEDIAN (A, B, x, y)

Step 1:  (1.1) low_A ← 1
         (1.2) low_B ← 1
         (1.3) high_A ← r
         (1.4) high_B ← s
         (1.5) n_A ← r
         (1.6) n_B ← s

Step 2:  while n_A > 1 and n_B > 1 do
         (2.1) u ← low_A + ⌈(high_A − low_A − 1)/2⌉
         (2.2) v ← low_B + ⌈(high_B − low_B − 1)/2⌉
         (2.3) w ← min(⌊n_A/2⌋, ⌊n_B/2⌋)
         (2.4) n_A ← n_A − w
         (2.5) n_B ← n_B − w
         (2.6) if a_u ≥ b_v
                  then (i) high_A ← high_A − w
                       (ii) low_B ← low_B + w
                  else (i) low_A ← low_A + w
                       (ii) high_B ← high_B − w
                  end if
         end while
```

Sa n_A i n_B označen je broj elemenata niza A i B koji se još uvijek uzimaju u obzir. Sa w označen je manji od $\lfloor n_A/2 \rfloor$ and $\lfloor n_B/2 \rfloor$. Medijani a i b od elemenata koji se još uvijek uzimaju u obzir u nizu A i B se upoređuju. Ako je a ≥ b tada najveći (najmanji) w elementi niza A (B) se uklanjaju u daljem razmatranju. Inače ako je a < b tada najmanji (najveći) w elementi niza A (B) se uklanjaju u daljem razmatranju. Proces se ponavalja sve dok ne ostane jedan element koji se razmatra u jednoj ili u obe od dve sekvence. Procedura prati elemente koji se uzimaju u obzir koristeći dva pokazivača low i high.

Složenost algoritma je O(log n) gde je n = r+s.

- **Fast merging on the EREW model**

Koristi se procedura TWO-SEQUENCE MEDIAN za konstruisanje paralelnog algoritma za EREW model.

Neka su data dva niza A=(a1, a2, a3...) i B=(b1,b2,b3...). Nizovi su sortirani. Algoritam podrazumijeva postojanje N procesora P1,P2,...PN, gde je N stepen dvojke i $1 \leqslant N \leqslant r + s$. Algoritam spaja nizove A i B u sortirani niz C u dve etape:

Korak 1: Svaki od dva niza A i B se deli u N podnizova A1,A2,...AN i B1,B2,...BN

a) $|A_i| + |B_i| = (r + s)/N$ for $1 \leqslant i \leqslant N$ i

b) Svi elementi u $A_i.B_i$ su manji ili jednaki od svih elemenata u $A_{i+1}.B_{i+1}$ za $1 \leqslant i \leqslant N$.

Korak 2: Svi parovi A_i i B_i, $1 \leqslant i \leqslant N$., se spajaju istovremeno i smještaju u niz C.

Prvi korak može da se implementira efikasno uz pomoć procedure TWO-SEQUENCE MEDIAN. Korak 2 koristi proceduru SEQUENTIAL MERGE. U sledećoj proceduri $A[i, j]$ se koristi da označi podsekvencu $\{a_i, a_{i+1}, \ldots, a_j\}$ niza A ako je i<= j inače $A[i, j]$ je prazan. Slično definišemo $B[i,j]$.

procedure EREW MERGE (A, B, C)

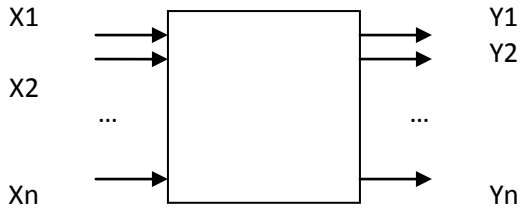
Step 1: (1.1) Processor P_1 obtains the quadruple (1, r, 1, s)
(1.2) for $j = 1$ to log N do
for i = 1 to 2^{j-1} do in parallel
Processor P_i having received the quadruple (e, f, g, h)
(1.2.1) {Finds the median pair of two sequences)
TWO-SEQUENCE MEDIAN $(A[e, f], B[g, h], x, y)$

(1.2.2) {Computes four pointers p_1, p_2, q_1, and q_2 as follows:}
if a, is the median
then (i) $p_1 \leftarrow x$
(ii) $q_1 \leftarrow x + 1$
(iii) if $b_y \leqslant a$, then (a) $p_2 \leftarrow y$
(b) $q_2 \leftarrow y + 1$
else (a) $p_2 \leftarrow y - 1$
(b) $q_2 \leftarrow y$
end if
else (i) $p_2 \leftarrow y$
(ii) $q_2 \leftarrow y + 1$
(iii) if $a_x \leqslant b_y$ then (a) $p_1 \leftarrow x$
(b) $q_1 \leftarrow x + 1$
else (a) $p_1 \leftarrow x - 1$
(b) $q_1 \leftarrow x$
end if
end if
(1.2.3) Communicates the quadruple (e, p_1, g, p_2) to P_{2i-1}
(1.2.4) Communicates the quadruple (q_1, f, q_2, h) to P_{2i}
end for
end for.

Step 2: for i = 1 to N do in **parallel**
Processor P_i having received the quadruple (a, b, c, d)
(2.1) $w \leftarrow 1 + ((i - 1)(r + s))/N$
(2.2) $z \leftarrow \min\{i(r + s)/N, (r + s)\}$
(2.3) SEQUENTIAL MERGE $(A[a, b], B[c, d], C[w, z])$
end for. □

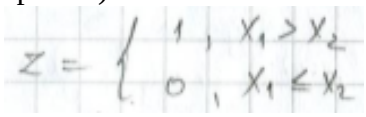
- T(s+r) = O(log N)* O (log min(r,s)) + O((r+s)/N)
  r=s=n
  T(2n) = O(logN*logn + n/N)
  C(2n) = O(NlogNlogn + n)
  NlogN * logn <= n
  logN < logn
  NlogN * logn < $Nlog^2$ n
  $Nlog^2 n<n$
  $N<n/log^2 n$
  C(n)=O(n)

- **Mreže za sortiranje**
  - To je jedno kombinatorno kolo sa n ulaza i n izlaza pri čemu su izlazi sortirani ulazi. Može se pretpostaviti:

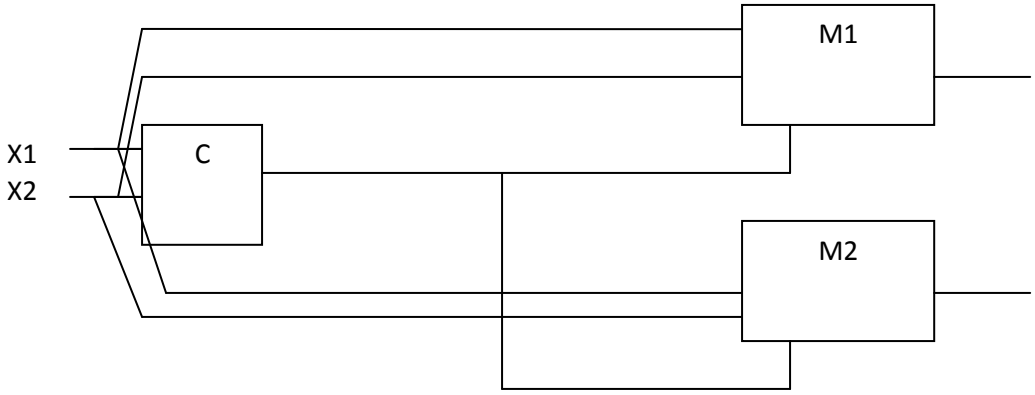


  Pretpostavimo da je na vrhu min, a na dnu max.
  - Mreža za sortiranje od 2 elementa:

  Ovu mrežu za sortiranje pravimo pomoću kombinatornih kola. Trebaju nam 2 MUX i jedan komparator(daje izlaz 0 ako je X_2>, a izlaz 1 ako je X_1> kad ih uporedi).

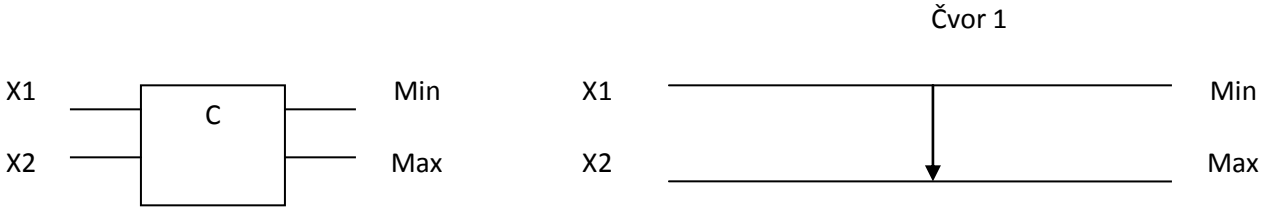


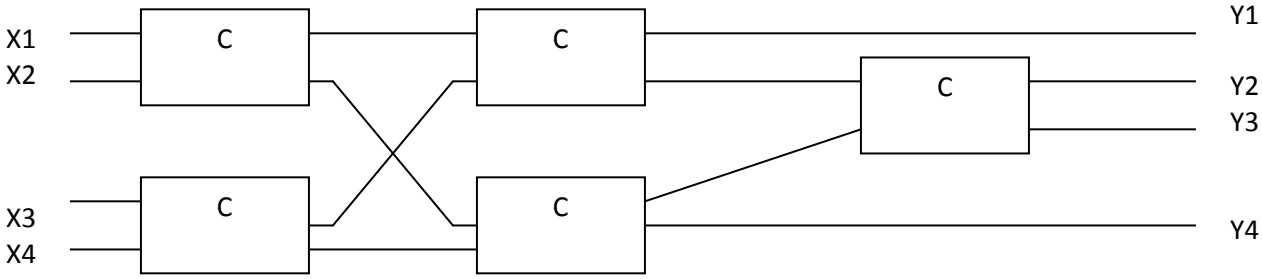
  MUX propušta jednu od mogućih vrijednosti.



Ako je na ulazu 0 propušta prvu prijednost.

- Kako realizovati mrežu sa n ulaza i n izlaza – koristimo komparatore 2x2:



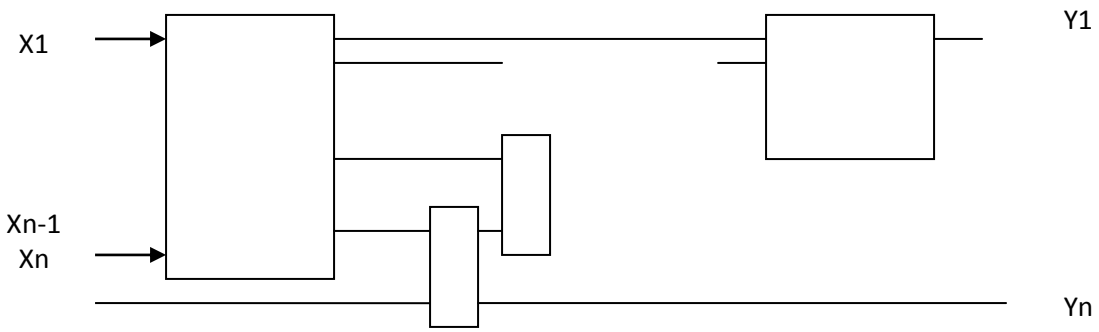


Prolaz kroz jedan element 2x2 zahtijeva konstantno vrijeme. Koji je najveći broj elemenata kroz koji prođe signal?

- Cijena=5 (br. elem. Komparatora); čekanje=3

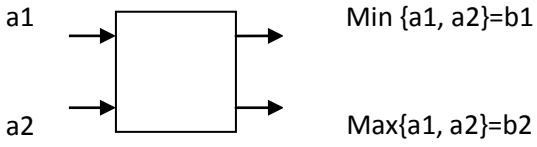
Prvo realizujemo mrežu sa n-1 elemenata, pa naknadno dodamo n-ti element. (Bazirano je na insert alrgoritmu)

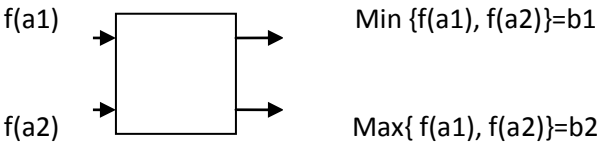
- I koliko koristimo elemenata
- II čekanje (vrijeme za sortiranje)



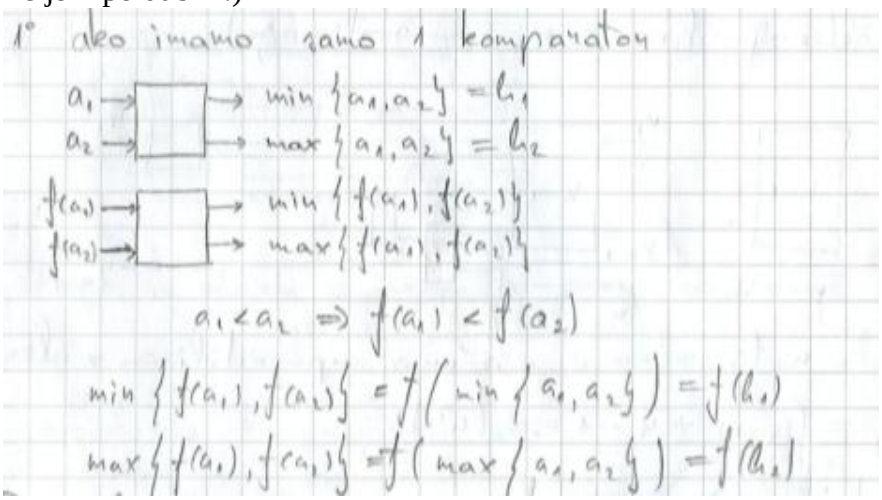
- $C_{(n)}=C_{(n-1)}+n-1=O(n^2)$
- Zadržavanje kroz ovu mrežu je
  $D_{(n)}=D_{(n-1)}+n-1= O(n^2)$

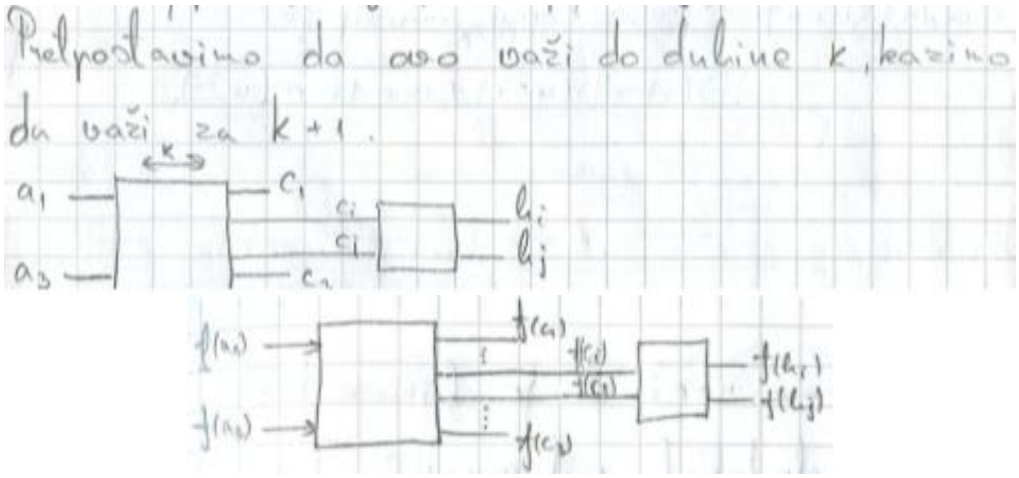
o Mjera da li koristimo mrežu 2x2:

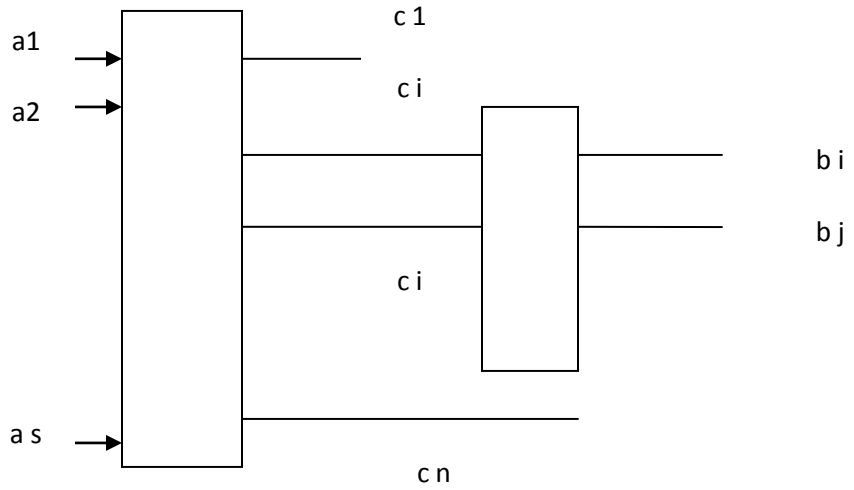




o **Teorema.** Neka je data monotono rastuća funkcija f. Tada ako mreža sastavljena od komparatora ima za ulaz $a_1 \ldots a_n$ daje izlaz $b_1 \ldots b_n$ onda će ta mreza za ulaz $f(a_1) \ldots f(a_n)$ dati izlaz $f(b_1) \ldots f(b_n)$.
D.(indukcijom po dubini.)







- **Teorema:** Neka imamo mrežu komparatora veličine nxn, tada ova mreža sortira proizvoljni niz brojeva $X_{1\ldots}X_n$ akko za sve moguće nizove 0 i 1 dužine n mreža ih sortira.

  (svaki ($y_1,\ldots,y_n/2^n$, y_i pripada $\{0,1\}$) mreža će sortirati ovaj niz.

  Ova teorema se zove 0-1 teorema jer proizvoljni niz prevodimo u niz 0 i 1.

  Dokaz: Pretpostavimo stuprotno, tj. da postoji niz $X_{1\ldots}X_n$ na kome mreza daje izlaz $a_1\ldots a_n$ pri čemu postoji i tako da je $a_i > a_{i+1.}$ Konstruišimo niz X_j koji je jednak:

$$z_j = f(x_j) = \begin{cases} 0, & x_j < a_i \\ 1, & x_j \geq a_i \end{cases}$$

  tako da je funkcija f definisana na ovaj način. Tada za ulaz z_j dobićemo i izlaz $f(a_j)$ $z_j=f(x_j) \rightarrow b_j = t(a_j)$, a pri tome važi:

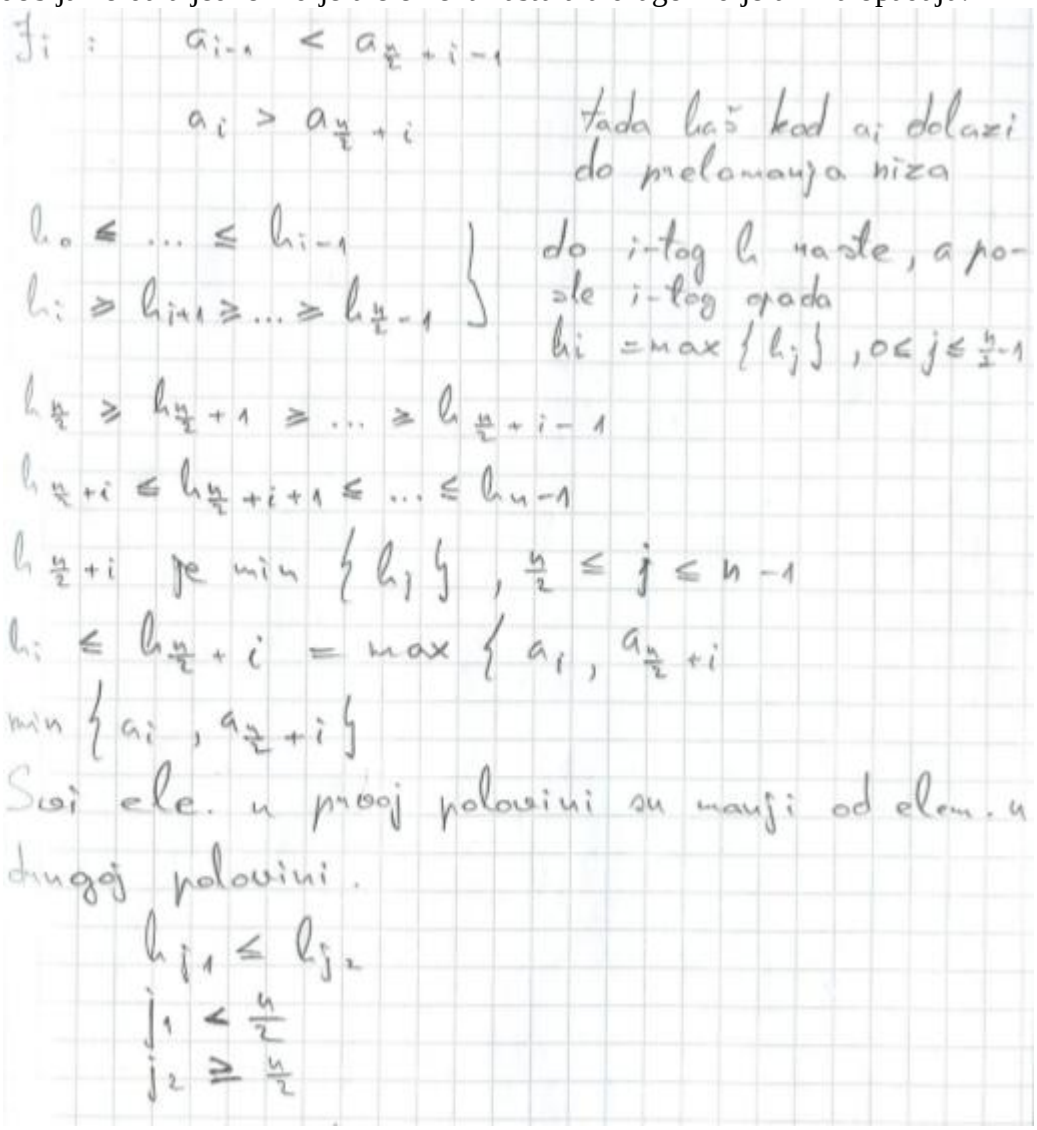
$$b_i = f(a_i) > f(a_{i+1}) = b_{i+1}$$

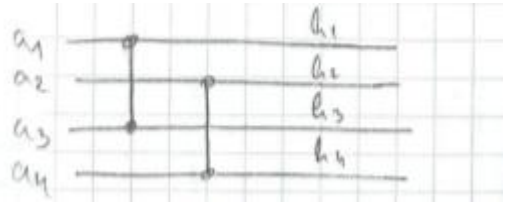
- Za niz $X_{1...}X_n$ ćemo reći da je **bimonoton** ako važi jedan od sledećih uslova:
  - Postoji i tako da $X_1 \leq \ldots \leq X_i$ i $X_{i+1} \geq X_{i+2} \geq \ldots \geq X_n$
  - Postoji i tako da $X_1 \geq \ldots \geq X_i$ i $X_{i+1} \leq X_{i+2} \leq \ldots \leq X_n$
  - Ako cikličnom permutacijom naših elementa možemo dobiti da važi 1 ili 2.

  Primjer. Neka imamo niz brojeva: 15,11,3,5,7,18,16. Da li je ovaj niz bimonoton? Odgovor je potvrdan jer cikličnom permutacijom utvrđujemo da raste do 18, pa opet pada:

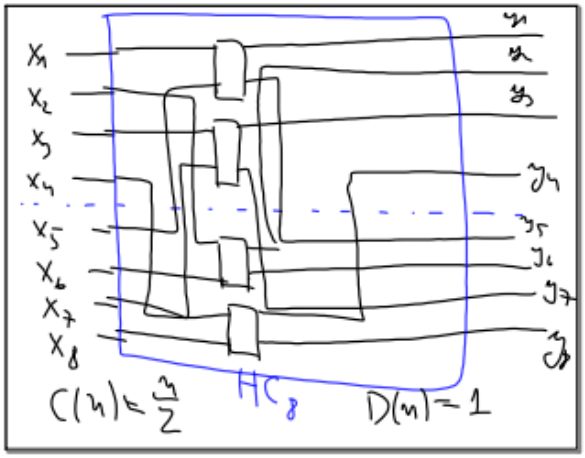
  3,5,7 ,**18**, 16,15,11

  Interesuje nas da napravimo šemu koja će vršiti sortiranje niza koji je bimonoton. Zamislimo da je $a_0 \ldots a_n$, $n = 2^k$ bimonoton niz.
  Definišimo niz $b_i = \min \{a_i, a_{n/2}\}$ pri čemu je $i<n/2$, $n=2^k$
  Ako pretpostavimo da je $a_0 \leq \ldots \leq a_{n/2}-1$ i $a_{n/2} \geq a_{n/2} \geq \ldots \geq a_{n-1}$,
  dobijamo da u jednom dijelu elementi rastu a u drugom dijelu niza opadaju.

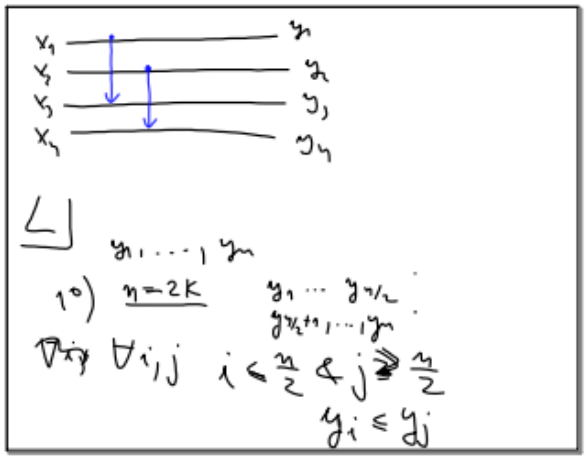




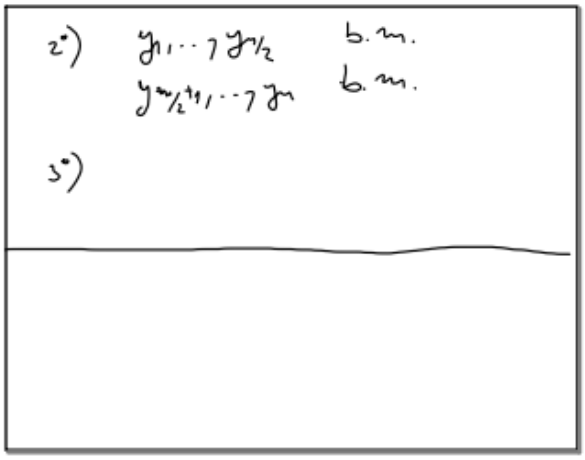
Dodatak mrežama za sortiranje (sa profesorskog računara)

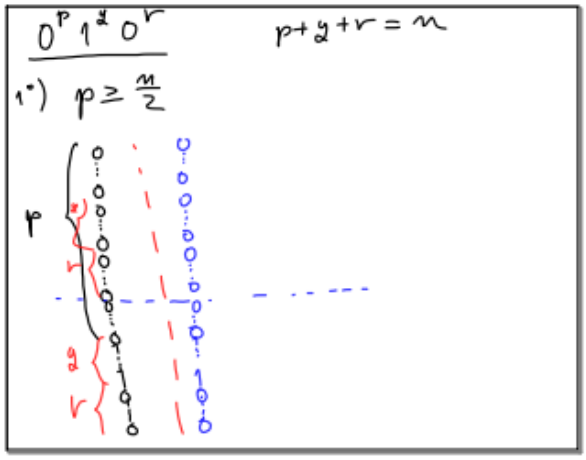


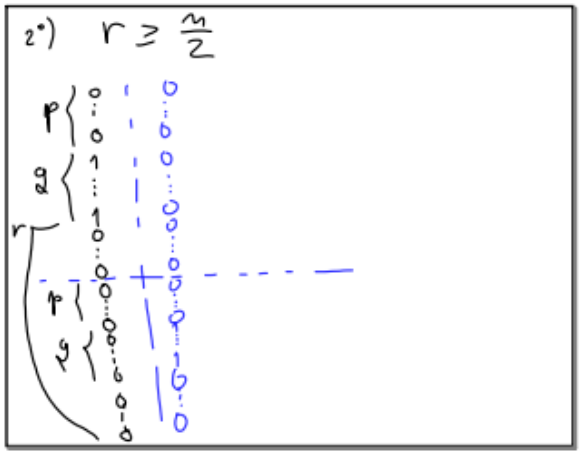
Nov 25-8:51 PM

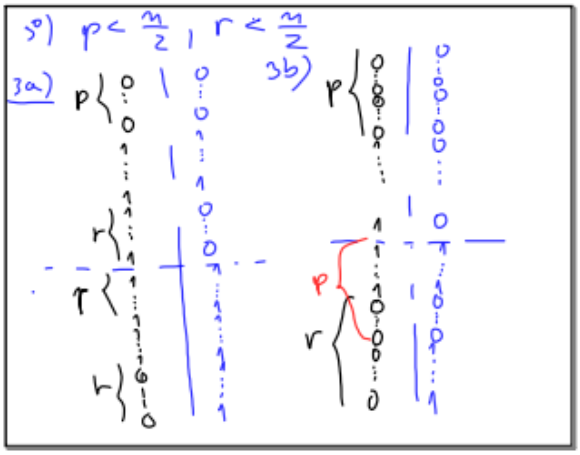


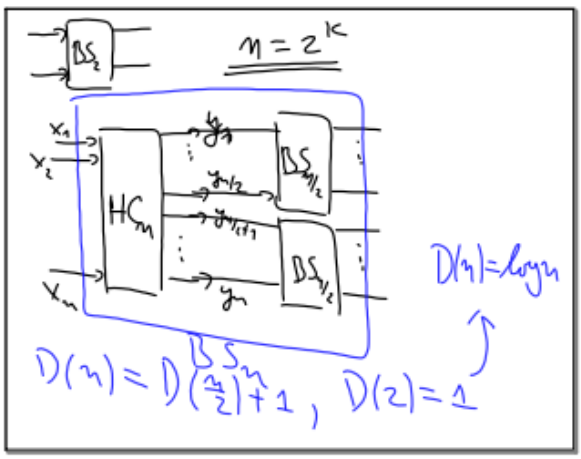
Nov 25-8:55 PM









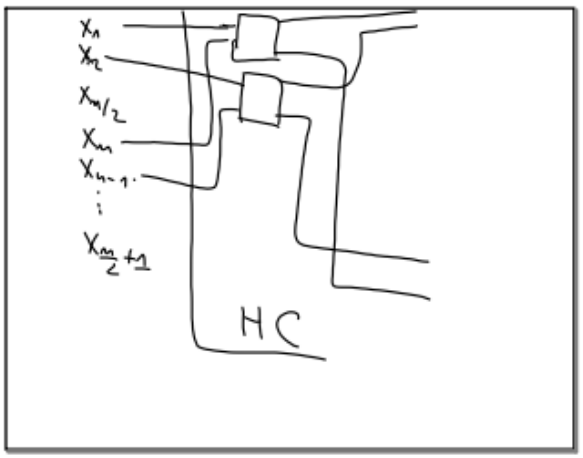


Nov 25-9:21 PM

$$C(n) = 2 \cdot C\left(\frac{n}{2}\right) + \frac{n}{2} \Rightarrow$$

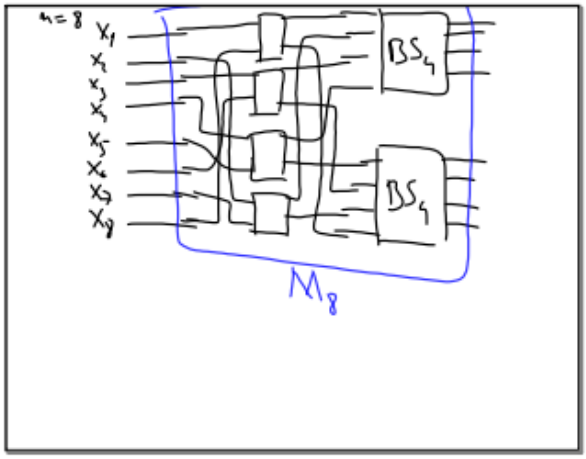
$$\Rightarrow C(n) = O(n \log n)$$

Nov 25-9:27 PM

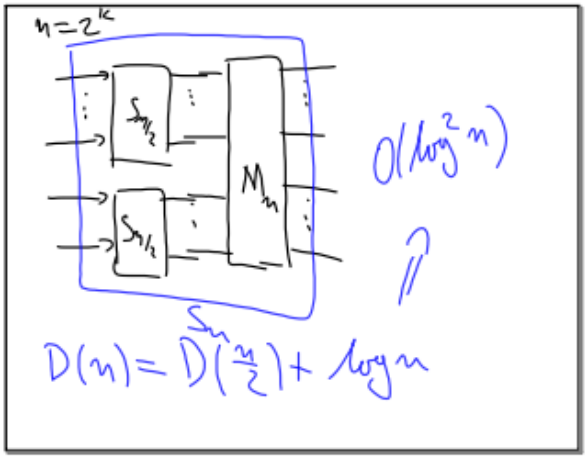


$$y_i = \min\{x_i, x_{n+1-i}\}$$

$$y_{n+1-i} = \max\{x_i, x_{n+1-i}\}$$



Nov 25-9:34 PM



Nov 25-9:37 PM

$C(n) = 2 \cdot C(\frac{n}{2}) + O(n \log n)$

$D(n) = O(\log n)$

$C(n) = O(n^2)$

Dodatak 2: Dokaz 0-1 teoreme (na engleskom)

An indispensable tool for the proof of correctness of sorting networks is the 0-1-principle [Knu 73]. The 0-1-principle states the following:

Theorem: (0-1-principle)

If a sorting network sorts every sequence of 0's and 1's, then it sorts every arbitrary sequence of values.

The proof of the 0-1-principle is not very difficult. However, it is quite helpful to have some definitions and lemmas ready.

Preliminaries

Definition: Let A and B be ordered sets. A mapping $f : A \rightarrow B$ is called monotonic if for all $a_1, a_2 \in A$

$$a_1 \leq a_2 \quad \Rightarrow \quad f(a_1) \leq f(a_2)$$

Lemma: Let $f : A \rightarrow B$ be a monotonic mapping. Then the following holds for all $a_1, a_2 \in A$:

$$f(\min(a_1, a_2)) = \min(f(a_1), f(a_2))$$

Proof: Let $a_1 \leq a_2$ and thus $f(a_1) \leq f(a_2)$. Then

$$\min(a_1, a_2) = a_1 \quad \text{and} \quad \min(f(a_1), f(a_2)) = f(a_1)$$

This implies

$$f(\min(a_1, a_2)) = f(a_1) = \min(f(a_1), f(a_2))$$

Similarly, if $a_2 \leq a_1$ and therefore $f(a_2) \leq f(a_1)$, we have

$$f(\min(a_1, a_2)) = f(a_2) = \min(f(a_1), f(a_2))$$

An analogous property holds for the max-function.

Definition: Let $f : A \rightarrow B$ be a mapping. The extension of f to finite sequences $a = a_0, ..., a_{n-1}$, $a_i \in A$ is defined as follows:

$$f(a_0, ..., a_{n-1}) = f(a_0), ..., f(a_{n-1}) , \quad \text{i.e.}$$

$$f(a)_i = f(a_i)$$

Lemma: Let f be a monotonic mapping and N a comparator network. Then N and f commutate, i.e. for every finite sequence $a = a_0, ..., a_{n-1}$ the following holds:

$$N(f(a)) = f(N(a))$$

In other words: a monotonic mapping f can be applied to the input sequence of comparator network N or to the output sequence, the result is the same.

Proof: For a single comparator $[i:j]$ the following holds (see definition of comparator):

$$[i:j](f(a))_i = [i:j](f(a_0), ..., f(a_{n-1}))_i = \min(f(a_i), f(a_j))$$

$$= f(\min(a_i, a_j)) = f([i:j](a)_i) = f([i:j](a))_i$$

This means that the ith element is the same regardless of the order of application of f and $[i:j]$. The same can be shown for the jth element and for all other elements. Therefore

$$[i:j](f(a)) = f([i:j](a))$$

For an arbitrary comparator network N (which is a composition of comparators) and a monotonic mapping f we have therefore

$$N(f(a)) = f(N(a))$$

Proof of the 0-1-principle

Theorem: (0-1-principle)

Let N be a comparator network. If every 0-1-sequence is sorted by N, then every arbitrary sequence is sorted by N.

Proof: Suppose a with $a_i \in A$ is an arbitrary sequence which is not sorted by N. This means $N(a) = b$ is unsorted, i.e. there is a position k such that $b_k > b_{k+1}$.

Now define a mapping $f : A \to \{0, 1\}$ as follows. For all $c \in A$ let

$$f(c) = \begin{cases} 0 & \text{if } c < b_k \\ 1 & \text{if } c \geq b_k \end{cases}$$

Obviously, f is monotonic. Moreover we have:

$$f(b_k) = 1 \quad \text{and} \quad f(b_{k+1}) = 0$$

i.e. $f(b) = f(N(a))$ is unsorted.

This means that $N(f(a))$ is unsorted or, in other words, that the 0-1-sequence $f(a)$ is not sorted by the comparator network N.

We have shown that, if there is an arbitrary sequence a that is not sorted by N, then there is a 0-1-sequence $f(a)$ that is not sorted by N.

Equivalently, if there is no 0-1-sequence that is not sorted by N, then there can be no sequence a whatsoever that is not sorted by N.

Equivalently again, if all 0-1-sequences are sorted by N, then all arbitrary sequences are sorted by N.