

SORTIRANJE1. Formulacija problema

Zasto sortiramo?

Zato da brže tražimo. Traženje u sortiranom nizu logaritamski ovisi o dužini niza (u najgorem slučaju), dok traženje u nesortiranom nizu ovisi linearno o dužini niza i u prosjeku (naravno, i u najgorem slučaju).

Model sortiranja:

Pretpostavljamo da su objekti koje treba sortirati zapravo recordi - zapisi, koji imaju više polja - rubrika.

Jedno od polja je takvog tipa da je za taj tip definirana relacija linearnog uređaja koju označavamo \leq . To polje zovemo ključ - key.

Naš zadatak je: zadani niz (konacni) od n takvih objekata poredati ulazno (ili izlazno) po mjeduošći rubrike ključ.

- Za zadane recorde r_1, \dots, r_n s mjeduošćima ključeva

$$k_1, \dots, k_n,$$

respektivno, treba poredati te recorde u redosljednu

$$r_{i_1}, \dots, r_{i_n}$$

talob da za pripadne ključeve vrijedi

$$k_{i_1} \leq k_{i_2} \leq \dots \leq k_{i_n}.$$

- Općenito - recorda s istom mjeduošću ključa može biti i više, tj. ne tražimo da sve mjeduošći ključeva budu različite.

Talodter, recordi s istom mjeduošću ključa mogu imati bilo koji međusobni poredak u uređenom nizu. Tj. osim po ključu, nema dodatnih zahtjeva na poredak recorda (na pr. sekundarnih ključeva i sl.).

Dodatna ograničenja i pretpostavke:

— trenutno promatramo samo problem tzv. internog (unutarnjeg) sortiranja. To znači da pretp. da je pristup ob. bilo kojeg recorda elementarna operacija - tj. ne ovisi o položaju recorda u nizu.

Model je koristan za sortiranje u glavnoj memoriji ili RAM-u - slučajni pristup podacima.

- Drugačiji model je tzv. eksterno (vanjsko) sortiranje - kada mjeme pristupa ovisi o položaju recorda u nizu.

Taj model odgovara sortiranju velikih nizova podataka na vanjskim memorijskim (disk, traka) - pa je medij sa slučajnim ili blok-slučajnim pristupom.

— Dodatno zahtijevamo da u postupku sortiranja konstantno najviše konstantnu količinu dodatne memorije - prostora, pored prostora za polazni niz.

To znači da sortiranje treba obaviti "u mjestu", zamjenama objekata.

Uz ova ograničenja, možemo specificirati strukturu podataka na kojoj radimo sortiranje:

```
type object = record
  key : key type; { linearni tip }
  info : bilo-što; { može : hrpa
  rubrika }
end;
```

Ovaj tip opisuje objekte - recorde koje treba sortirati po komponenti key. Ostatak sadržaja, koji smo nazvali info - ne gledamo.

Taj dio namu ulazi

Za sortiranje možemo (i moramo) koristiti slijedeće dvije operacije - koje smatramo elementarnima:

(a) uspoređivanje ključeva - preciznije, provjeru relacije $key_1 < (ili \leq) key_2$, s rezultatom tipa boolean.

Ovo je, obično, brza operacija, ako je "linearni-tip" ueti od osnovnih tipova, poput integer, char, real.

Ta operacija može biti i nešto sponja, ako je ključ na pr. polje znakova ili string, s leksikografskim uređajem.

(b) seljenje - kopiranje ili pomak cijelog objekta (recorda).

(kopiranje = dodjeljivanje vrijednosti tj. operator := za tip object).

Ta operacija je, također, elementarna, po našoj pretp., ali može imati bitno različito trajanje od uspoređivanja. Velikina komponente info direktno određuje trajanje kopiranja.

To je i jedino mjesto u sortiranju, gdje se pojavljuje utjecaj "info" komponente.

To znači da u analizi složenosti algoritama sortiranja treba posebno brojati ove 2 vrste operacija (jer ne moraju podjednako trajati; iako su tipa $O(1)$ - tj. konstantne!).

- Za interno sortiranje, možemo pretpostaviti da je niz recorda reprezentiran poljem recorda (indeksiranje, po trajanju, ne ovise o indeksu!)

type index = 1..nmax; { nmax = konstanta }
 polje = array [index] of object;

Konечно je dozvoliti da duljina uiza ne mora biti točno n_{max} , nego i manja.

Zbog toga uveli smo varijablu (ili konstantu)

var n : index;

koja sadrži stvarnu duljinu uiza, a smatramo da se uiz nalazi na prvih n mjestima (indeksa) u polju.

- Zbog ograničenja na dodatni prostor, možemo pretpostaviti da se sve odvija unutar istog polja

var A : polje;

tj. $A[i] = r_i, i=1, \dots, n$
na poč.

koje, na početku, sadrži polazni uiz recorda (na prvih n mjestima), a na izlazu sadrži poredani uiz tih recorda (opet na prvih n mjestima)

- Dalje, sortiranje uziho promijenama poretka (zamjenama sadržaja) objekata u polju A .

Napomena: Iz analize algoritama sledi da za broj kopiranja recorda imajedi $O(n)$ (bar za neke algoritme), tj. nema biti potrebe za dodatnim poljima!

Napomena: Kopiranje recorda se mogu potpuno izbjeći, ako ne zahtjevamo baš strogo preuređenje uiza.

Dovoljno bi bilo naći permutaciju $p \in S_n$

$$p(j) = i_j, \quad j=1, \dots, n$$

(tačnik može biti i više, ako ima istih ključeva) koja kaže kako treba preurediti uiz,

$$r_1, r_2, \dots, r_n \rightarrow r_{p(1)}, r_{p(2)}, \dots, r_{p(n)}.$$

No, paucenje permutacije p zahtjeva još jedno polje
var p : array [index] of index.

Osim toga, kod pretraživanja stalno treba konstiti permutirane indekse - tj. dvostruki indeks (indeks od indeksa) - što usporava pretragu.

Naime, ideja sortiranja je da se traženje obavlja mnogo puta - tako da treba strahiti uopće ništa.

```
{*****}
      SORTIRANJE - Minimum Exchange Sort algoritam.
{*****}

procedure Min_Sort ( n : index ; var A : polje ) ;
    { Sortira uzlazno elemente A[1], ..., A[n] u polju A. }

var
    i, j : index ;
    min_ind : index ;
    min_key : key_type ;
    tmp_obj : object_type ;

begin
    for i := 1 to n - 1 do
        begin
            min_key := A[i].key ;
            min_ind := i ;
            for j := i + 1 to n do
                if A[j].key < min_key then
                    begin
                        min_key := A[j].key ;
                        min_ind := j ;
                    end ; { if, for j }
            if min_ind <> i then
                begin
                    tmp_obj := A[i] ;
                    A[i] := A[min_ind] ;
                    A[min_ind] := tmp_obj ;
                end ; { if min_ind <> i }
            end ; { for i }
        end ; { Min_Sort }

{*****}
□
```

3. Quicksort algoritam

Originalni algoritam: C.A.R. Hoare (1962)
uz nekoliko kasnijih poboljšanja.

- To je, u prosjeku - empirijski, najefikasniji poznati alg. sortiranja (bez strogo dozra!).

- Osnovna ideja je rekurzivna, vrlo ualaz na raspolaganje odn. binarno pretrazivanje:

- izaberemo neki ključ - tzv. pivotni ključ v_k oko kojeg ćemo raspodijeliti polje na 2 dijela.

- preuredimo (permutiramo) polje, tako da za neki indeks $j \in \{1, \dots, n\}$ vrijedi:

$$\begin{aligned} A[i].key < v_k & \text{ za } i < j, & (\text{može } \leq, i \leq k) \\ A[i].key > v_k & \text{ za } i > j, & (\text{može } >, i > k) \end{aligned}$$

- rekurzivno sortiramo 2 manja polja:

$$A[1..j-1] \text{ i } A[j..n].$$

- Pri tome se nadamo da je izabrani pivotni ključ u blizu mediana - polovišta, tj. da je $k \approx \frac{n}{2}$, tako da su manja polja podjednake duljine

- Za rekurziju treba parametrizirati ulaz tako da se lako zadaje potpolje koje treba sortirati tj. treba zadati polazni i krajnji indeks u potpolju:

$$A[i..j]$$

- Drugim riječima, razradjemo (ili projektiramo) potprogram quicksort:

procedure quicksort (i, j : integer);

koja sortira elemente $A[i], \dots, A[j]$ globalnog polja A.

- Oprez - odmah treba uočiti da moramo paziti na to da na ulazu vrijedi $i \leq j$.

Druga varijanta - ako je $i > j$ - ništa ne radimo. Također: ako je $i = j \rightarrow$ ništa ne radimo, jer je potpolje od 1 elementa uvijek sortirano.

Treba paziti na to da zaista dobijemo 2 najmaja polja, a ne jedno.

Naime, ako za v^k slučajno izaberemo najmanji ključ u polju, onda je potpolje strogo manjih od njega

$$A[i..k-1] \quad k = j-1$$

prazno, jer je $k = i$ (1)

Tada ostajemo sa samo jednim potpoljem - onih koji su veći ili jednaki od v^k ,

$$A[k..j] = A \quad (\text{jer } k=i)$$

a to je, opet, citavo polazno polje $\Rightarrow \infty$ proces. (nema smanjenja velicine).

- To se lako ujedna: potpolje strogo manjih je korektno definirano, ako i samo ako imamo bar 2 razlicita kljuc̃a i v^k nije najmanji (ima bar jedan strogo manji).

- Dakle, kad trazimo v^k - treba provjeriti da li u trenutnom potpolju postoje bar 2 razlicita kljuc̃a i v^k veći od ta 2.

U protivnom - svi kljuc̃evi su isti, pa ne treba sortirati.

- Ovaj posao obavlja procedura findpivot:

```
procedure findpivot (l, r: integer;
var ok: boolean; {= našao pivot}
var pivotindex: index
{ indeks pivotnog elementa u polju });
```

učen posao: ako je $i \neq j$ (!) i firstkey := $A[i].key$ i u komadu polja $A[i+1..j]$ bari prvi ključ različit od firstkey. ako ga nađeš, pivotni indeks je indeks većeg od ta 2 ključa.

Potpuni potprogram za Quicksort algoritam:

```
procedure Qsort ( n: index;
                  var A: polje );
  { Sortira uzlazno elemente A[1], ..., A[n] polja A }
```

```
procedure findpivot ( i, j: integer;
                     var ok: boolean; { = našli pivot }
                     var pivotindex: index
                           { = indeks pivotnog elementa }
                     );
```

{ ok = u polju A[i..j] postoje bar 2 različita ključa. Tada je pivotindex = indeks većeg. Tražimo od i prema j. }

```
var firstkey: keytype;
    k: integer;
```

```
begin
```

```
  ok := false;
```

```
  if i < j then
```

```
    begin
```

```
      firstkey := A[i].key;
```

```
      k := i + 1;
```

```
      repeat
```

```
        if A[k].key > firstkey then
```

```
          begin
```

```
            ok := true; pivotindex := k;
```

```
          end
```

```
        else if A[k].key < firstkey then
```

```
          begin
```

```
            ok := true; pivotindex := i;
```

```
          end
```

```
        else
```

```
          k := k + 1;
```

```
        until ok or (k > j);
```

```
      end;
```

```
    end; { findpivot }
```

```

procedure partition ( i, j : integer;
                      pivotkey : keytype;
                      var k : integer );
{ Preuredite polje A[i..j] tako da je na izlazu:
  A[i], ..., A[k-1] < pivotkey
  A[k], ..., A[j] ≥ pivotkey. }
var l, r : integer;
begin
  l := i; r := j;
  repeat
    while A[l].key < pivotkey do l := l + 1;
    while A[r].key ≥ pivotkey do r := r - 1;
    if l < r then zamijeni (A[l], A[r]);
  until l > r;
  k := l;
end; { partition }

```

```

procedure quicksort ( i, j : integer );
{ Sortiraj uzlazno elemente A[i], ..., A[j]
  globalnog polja A }
var pivotkey : keytype;
    pivotindex, k : integer;
    ok : boolean;
begin
  findpivot ( i, j, ok, pivotindex );
  if ok then
    begin
      pivotkey := A[pivotindex].key;
      partition ( i, j, pivotkey, k );
      quicksort ( i, k - 1 );
      quicksort ( k, j );
    end;
  end; { quicksort }

begin { Qsort }
  quicksort ( 1, n );
end; { Qsort }

```

Složenost Quicksort algoritma

Pogledajmo prvo vremensku složenost u najgorem slučaju.

Neka je $T_Q(i, j)$ vrijeme potrebno za izvršavanje procedure Quicksort (i, j) . Quicksort zove pivot i partition na $A[i..j]$ i samog sebe na $A[i..k-1]$ i na $A[k..j]$

Tada je:

$$T_Q(i, j) \leq T_{\text{pivot}}(i, j) + T_{\text{partition}}(i, j) + c + T_Q(i, k-1) + T_Q(k, j)$$

Lako se vidi da su vremena za pivot i partition najviše linearna u duljini polja, jer se svaki element polja obrađuje najviše jednom, tj.

$$\begin{aligned} T_{\text{pivot}}(i, j) &\in O(j-i+1) & \text{ili } \leq c_1(j-i+1) \\ T_{\text{partition}}(i, j) &\in O(j-i+1) & \text{ili } \leq c_2(j-i+1) \end{aligned}$$

pa možemo pisati:

$$T_Q(i, j) \leq c'(j-i+1) + c + T_Q(i, k-1) + T_Q(k, j)$$

Dakle, bez rekurzivnih poziva, pojedini Quicksort (i, j) traje linearno u duljini polja na kojemu radi (lin. u broju elem. koji se sortira).

Zbog toga je lakše pisati T_Q kao funkciju duljine polja:

$$T_Q(i, j) \rightarrow T_Q(j-i+1).$$

Naš polazni zadatak je sortiranje polja duljine n . Tada je:

$$T_Q(n) \leq T_Q(k-1) + T_Q(n-k+1) + c'n + c.$$

- Ako imamo sreću, pa je stalno $k \approx \frac{n}{2}$, dobivamo rekurzivni oblika

$$T_Q(n) \leq 2T_Q\left(\frac{n}{2}\right) + c'n + c$$

→ rješenjem: $T_Q(n) = O(n \log n)$.

- U najgorom slučaju je $k=2$ ili n , tj. potpolja su duljine 1 i $n-1$.

Dato je $T_Q(1) \leq c''$, pa izlazi:

$$T_Q(n) \leq T_Q(n-1) + c \cdot n + c + c''$$

s ožetjenjem: $T_Q(n) = O(n^2)$.

- Lako se vidi da je i u donjoj ogradi

$$T_Q(n) = \Omega(n \log n)$$

jer svaki poziv procedure (bilo koje) ponešto traje. Ovo mijedi ako su svi ključevi međusobno različiti (u slučaju pivot uvoje vrabiti vrijeme na $O(n)$).

- Najgora složenost se može dostići. Primer je naopako sortirano polje (silazno poredano). Tada je pivotni ključ uvijek najveći element polja, tj. desno potpolje je uvijek duljine 1.

- Što se broja usporedbi tiče, mijedi potpuno ista argumentacija (svaki element sudjeluje u najviše 2 usporedbe u pivotu i partition).

- Zamjena se javlja samo u partition. Opet, može se desiti - naopako sortirano polje - da svaki element polja mora promijeniti mjesto.

Tj. broj zamjena može biti linearan u duljini polja (unutar partition), tj. globalno dobijemo za ukupni broj zamjena: $N_{\text{zamjena}}(n) = O(n^2)$.

- Srećom, u praksi, dobivamo mnogo bolju ocjenu.

Prosječna vremenska složenost Quicksorta

Za prosječnu složenost uovamo ponešto pretpostaviti o distribuciji ulaza, i definirati prosj. složenost.

Polazne pretpostavke:

- (1) - svi polazni poreci (redoslijedi - permutacije) su jednako vjerojatni,
 - nema jednakih ključeva (ovo je samo zbog pojednostavljenja)
- Naime, isti ključevi samo ubrzavaju algoritam jer se taj dio, zapravo, ne sortira.

To znači da imamo $n!$ jednako vjerojatnih poredaka na ulazu. Prosječnu vrijednost gledamo kao srednju vrijednost po svim polaznim porecima. (perm.)

- Za pojednostavljenje analize, uvodimo još jednu pretpostavku:

- u trenutku kad pozivamo $\text{quicksort}(i, j)$ na $A[i..j]$, svi poreci za $A[i], \dots, A[j]$ su, također, jednako vjerojatni.
- (ovo je mnogo strože! Ranija pretp. je ovo isto, ali samo za poziv $\text{quicksort}(1, n)$).

Opravdanje: (umjesto strogoj dožazi)

Prije ovog poziva, svi raniji pivotni ključevi v nisu napravili particiju - rez u tom dijelu polja $A[i..j]$.

Tj. svi oni elementi su bili ili strogo manji od v ili su svi bili veći ili jednaki v .

(To je \Leftrightarrow kao da uzeli ovaj blok interpretiramo kao 1 element - bez pojedinačnog razlikovanja)

Preciznije: može se (pedantnom analizom) dožazati da bilo koji pivot ima veću šansu da završi blizu desnog ruba desnog potpolja (ovih većih ili jednakih od njega).

Međutim, za velika potpolja, ta činjenica da je minimum potpolja (tj. prošli pivot) vjerojatniji bliže desnom rubu, nema bitni značaj.

Neka je $T(n)$ prosječno vrijeme potrebno quicksortu za sortiranje polja od n elemenata.

Ideja - odozgo ograditi $T(n)$ u rekurzivnom obliku.

Start: Očito je $T(1) = c_1$ ($c_1 \leq c_1$) za neku konstantu $c_1 > 0$, jer za jedan jedini element nema rekurzije u quicksortu.

Pretpostavimo da je $n > 1$. Zbog pretpostavke da su svi ključevi različiti - postoje bar 2 različita, pa quicksort nalazi pivot i particionira polje. Taj dio posla (findpivot i partition) traje najviše

$$t(n) \leq c_2 n$$

vremena, za neku konstantu $c_2 > 0$ (aditivni dio uvlacimo u faktor c_2 , tj. umjesto $c_2 n + c$, pišemo $c_2 n$, jer $c > 0$).

Ova ograna vrijedi za bilo koji polazni poredak (kao gornja ograda), pa vrijedi i u prosjeku.

- Nakon toga, quicksort rekurzivno sortira 2 manja polja. Dakle:

$$T(n) \leq \underbrace{\text{prosječno trajanje rekurzije}}_{T_R(n)} + c_2 \cdot n.$$

- Treba naći prosječno trajanje rekurzije $T_R(n)$

Očito je:

$$T_R(n) = \sum_{i=1}^{n-1} p_i \cdot [T(i) + T(n-i)]$$

gdje je p_i vjerojatnost da lijevo potpolje ($<$ od pivota) ima točno i elemenata ($i \in \{1, \dots, n-1\}$ - jer sigurno imamo 2 potpolja).

Bilo bi lijepo kad bismo odmah mogli zaključiti da su sve veličine lijevog potpolja (od 1 do $n-1$) jednako vjerojatne tj. da je

$$p_1 = \dots = p_{n-1} = \frac{1}{n-1}.$$

Uočimo, da zbog pretp. da su svi slučajevi različiti, pivotni element mora biti najmanji element desnog potpolja. Tj. ako je lijevo potpolje duljine i , pivotni element ima poziciju točno $i+1$ u sortiranom poretku svih n elemenata.

- Za ulazenje p_i treba analizirati rad procedure findpivot - koja bira većeg od prva 2 elementa (u polaznom poretku).

- Pretpostavimo da je lijevo potpolje duljine i , tj. pivot je $i+1-i$ elem. po veličini.

- U našem izboru imamo 2 mogućnosti:

(1) pivot je na prvoj mjestu, a jedan od i elemenata manji od njega je na drugom mjestu.

Zbog pretp. o jednakoj vjerojatnosti svih ulaznih poredata, vjeroj. da je bilo koji određeni element (na pr. $i+1-i$ po redu veličini) na prvoj mjestu je

$$\frac{1}{n}.$$

Ako je on prvi, vjerojatnost da je drugi element jedan od i manjih od njega među $n-1$ preostalih je

$$\frac{i}{n-1}$$

Dakle, vjerojatnost da se ovaj slučaj dogodi je

$$\frac{i}{n(n-1)}$$

(2) pivot je na drugom mjestu, a jedan od i manjih od njega među $n-1$ preostalih je na prvom.

Potpuno analogno - vjerojatnost je ista

$$\frac{i}{n(n-1)}$$

- Vjerojatnost p_i dobivamo kao sumu ove dvije

$$p_i = \frac{2i}{n(n-1)}, \quad i \in \{1, \dots, n-1\}$$

jer (1) ili (2) daje element potpuno dužine i .

- Rekurentna za $T(n)$ ima oblik:

$$T(n) \leq \sum_{i=1}^{n-1} \frac{2i}{n(n-1)} [T(i) + T(n-i)] + c_2 n.$$

- Za pojednostavljenje konstruimo relaciju:

$$(*) \quad \sum_{i=1}^{n-1} f(i) = \frac{1}{2} \sum_{i=1}^{n-1} [f(i) + f(n-i)]$$

koja izlazi iz $\sum_{i=1}^{n-1} f(i) = \sum_{i=1}^{n-1} f(n-i)$ (zamjena indeksa sumacije $i \mapsto n-i$)

- Stavimo prvo $f(i) = \frac{2i}{n(n-1)} [T(i) + T(n-i)]$, izlazi:

$$T(n) \leq \frac{1}{2} \sum_{i=1}^{n-1} \left[\frac{2i}{n(n-1)} [T(i) + T(n-i)] + \frac{2(n-i)}{n(n-1)} [T(n-i) + T(i)] \right] + c_2 n.$$

Skratimo $\frac{1}{2}$ i 2 i uočimo: $\frac{i}{n} < 1$, $\frac{n-i}{n} < 1$

pa je:

$$T(n) \leq \frac{2}{n-1} \sum_{i=1}^{n-1} T(i) + c_2 n$$

↳ ovo je uvre i, n , ali ne treba!

- Po (*), ovo je rekurzija koju bismo dobili za slučaj $p_1 = \dots = p_{n-1} = \frac{1}{n-1}$ - jednakovjernoj. sve duljine lijevog potpolja.

Dalje, izbor većeg od 2 ključa ne utiče bitno na raspodjelu veličine potpolja.

- Ova rekurzija se uvre i direktno rješavati (sami).
- Nama je dovoljno dokazati da $\exists c > 0$ konst. takva da je $T(n) \leq c \cdot n \cdot \log n, \forall n \geq 2$

što dokazujemo indukcijom.

- Baza je za $n=2$. Tada je

$$T(2) \leq 2 \cdot T(1) + 2c_2 = (\text{ili } \leq) 2c_1 + 2c_2$$

pa iz $T(2) \leq c \cdot 2 \log 2$ slijedi

$$c_1 + c_2 \leq c \cdot \log 2$$

ili

$$c \geq \frac{c_1 + c_2}{\log 2}$$

Pretpostavimo da je c tako odabran i da vrijedi

$$T(i) \leq c \cdot i \cdot \log i, \quad i = 1, \dots, n-1$$

iz rekurzije je:

$$T(n) \leq \frac{2c}{n-1} \sum_{i=1}^{n-1} i \log i + c_2 n$$

Ovu sumu rastavljamo u 2 pojednaka dijela koje onda oženjivamo:

$$T(n) \leq \frac{2c}{n-1} \left[\sum_{i=1}^{n/2} i \log i + \sum_{i=n/2+1}^{n-1} i \log i \right] + c_2 n$$

$$\leq \begin{cases} \text{za } i=1, \dots, \frac{n}{2} \text{ je } \log i \leq \log \frac{n}{2} = \log n - 1 \quad \left(\begin{smallmatrix} \log \\ = \lg \end{smallmatrix} \right) \\ \text{za } i=\frac{n}{2}+1, \dots, n-1 \text{ je } \log i < \log n \end{cases}$$

$$\leq \frac{2c}{n-1} \left[\sum_{i=1}^{n/2} i (\log n - 1) + \sum_{i=n/2+1}^{n-1} i \log n \right] + c_2 n$$

$$\leq \left\{ \begin{array}{l} \sum_{i=1}^{n/2} i = \frac{1}{2} \cdot \frac{n}{2} \cdot \left(\frac{n}{2} + 1 \right) \\ \sum_{i=\frac{n}{2}+1}^{n-1} i = \frac{3}{4} n \left(\frac{n}{2} - 1 \right) \end{array} \right\}$$

$$\leq \frac{2c}{n-1} \left[\frac{n}{4} \left(\frac{n}{2} + 1 \right) \cdot \log n - \frac{n}{4} \left(\frac{n}{2} + 1 \right) + \frac{3}{4} n \left(\frac{n}{2} - 1 \right) \log n \right] + c_2 n$$

$\frac{\frac{n^2}{8} + \frac{n}{4}}{\quad \quad \quad} \quad \quad \quad \frac{\frac{3n^2}{8} - \frac{3n}{4}}{\quad \quad \quad}$

$$\leq \frac{2c}{n-1} \left[\left(\frac{\frac{n^2}{2} - \frac{n}{2}}{\frac{n(n-1)}{2}} \right) \log n - \left(\frac{\frac{n^2}{8} + \frac{n}{4}}{\quad} \right) \right] + c_2 n$$

$$\leq c \cdot n \log n - \frac{cn}{4} - \frac{cn}{2(n-1)} + c_2 n$$

Ako stavimo $c \geq 4c_2$ sama zadnja 3 člana je manja ili jednaka 0, pa je

$$T(n) \leq c \cdot n \cdot \log n.$$

```

{*****}
      SORTIRANJE - Quicksort (Knuth) algoritam.
{*****}

procedure Q_Sort ( n : index ; var A : polje ) ;
  { Sortira uzlazno elemente A[1], ..., A[n] u polju A. }
procedure Quicksort ( l, r : index ) ;
  { Sortira uzlazno elemente A[l], ..., A[r] u globalnom polju A. }

var
  pivot_key : key_type ;
  i, j, pivot_index : index ;
  tmp_obj : object_type ;

begin { Quicksort }
  i := 1 ;
  j := r + 1 ;
  {
  pivot_index := 1 ;
  }
  pivot_key := A[l].key ;
  repeat
    repeat
      i := i + 1 ;
    until A[i].key >= pivot_key ;
    repeat
      j := j - 1 ;
    until A[j].key <= pivot_key ;
    if i < j then
      begin
        tmp_obj := A[i] ;
        A[i] := A[j] ;
        A[j] := tmp_obj ;
      end ; { if i < j }
    until i >= j ;
    tmp_obj := A[l] ;
    A[l] := A[j] ;
    A[j] := tmp_obj ;
    if l < j - 1 then
      Quicksort ( l, j - 1 ) ;
    if j + 1 < r then
      Quicksort ( j + 1, r ) ;
  end ; { Quicksort }

begin { Q_Sort }
  if n > 1 then
    begin
      A[0].key := min_key ;
      A[n + 1].key := max_key ;
      Quicksort ( 1, n ) ;
    end ;
end ; { Q_Sort }

{*****}
□

```

```
{ SORTIRANJE - Quicksort (Wirth) algoritam. }

{*****}

procedure Q_Sort ( n : index ; var A : polje ) ;

    { Sortira uzlazno elemente A[1], ..., A[n] u polju A. }

procedure Quicksort ( l, r : index ) ;

    { Sortira uzlazno elemente A[l], ..., A[r] u globalnom polju A. }

var
    pivot_key : key_type ;
    i, j, pivot_index : index ;
    tmp_obj : object_type ;

begin { Quicksort }
    i := 1 ;
    j := r ;
    pivot_index := ( 1 + r ) div 2 ;
    pivot_key := A[pivot_index].key ;
    repeat
        while A[i].key < pivot_key do i := i + 1 ;
        while A[j].key > pivot_key do j := j - 1 ;
        if i <= j then
            begin
                tmp_obj := A[i] ;
                A[i] := A[j] ;
                A[j] := tmp_obj ;
                i := i + 1 ;
                j := j - 1 ;
            end ; { if i <= j }
    until i > j ;
    if l < j then Quicksort ( l, j ) ;
    if i < r then Quicksort ( i, r ) ;
end ; { Quicksort }

begin { Q_Sort }
    Quicksort ( 1, n ) ;
end ; { Q_Sort }

{*****}
```