SORTIRANJE

1. Formulacija problema

Zašto sortiramo?

Zato da brže tražimo. Traženje u sortiranom nizu logaritamski ovisi o duljini niza (u najgorem slučaju), dok traženje u nesortiranom nizu ovisi linearno o duljini niza i u prosjeku (usavršno, 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 $\leqslant$. To polje zovemo ključ - key.

Naš zadatak je: zadani niz (konačni) od n takvih objekata poredati uzlazno (ili silazno) po vrijednosti rubrike ključ.

- Za zadane recorde $r_1, \ldots, r_n$ s vrijednostima ključeva
$$k_1, \ldots, k_n,$$
respektivno, treba poredati te recorde u redoslijedu
$$r_{i_1}, \ldots, r_{i_n}$$
tako da za pripadne ključeve vrijedi
$$k_{i_1} \leqslant k_{i_2} \leqslant \ldots \leqslant k_{i_n}.$$

- Općenito - record s istom vrijednošću ključa može biti i više, tj. ne tražimo da sve vrijednosti ključeva budu različite.

Također, recordi s istom vrijednošću 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 do bilo kojeg recorda elementarna operacija - tj. ne ovisi o položaju recorda u nizu.

Model je korektan za sortiranje u glavnoj memoriji tj. RAM-u - slučajni pristup podacima.

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

Taj model odgovara sortiranju velikih nizova podataka na vanjskim memorijama (disk, traka) - pa je medij sa sekvencijalnim tj. blok-sekvencijalnim pristupom.

— Dodatno zahtijevamo da u postupku sortiranja koristimo 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 objekt = record
                key : keytype; {ključni tip}
                info : bilo_sto; } može i hrpa
              end;                 rubrika
```

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

Taj dio nama nije

Za sortiranje možemo (i moramo) koristiti sljedeće 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" neki od osnovnih tipova, poput integer, char, real.

Ta operacija može biti i nešto sporija, 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 načelu pretp., ali može trajati bitno različito trajanje od uspoređivanja. Veličina komponente zato direktno određuje trajanje kopiranja.

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

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

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

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

Korisno je dozvoliti da duljina niza ne mora biti točno nmax, nego i manja.

Zbog toga uvodimo varijablu (ili konstantu)

var n : index;

koja sadrži stvarnu duljinu niza, a smatramo da se niz nalazi na prvih n mjesta (indeksa) u polju.

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

var A : polje;

koje, na početku, sadrži polazni niz records (na prvih n mjesta), a na izlazu sadrži poredani niz tih records (opet na prvih n mjesta) tj. $A[i] = r_i$, $i = 1, \dots, n$ na pč.

- Dalje, sortiranje vrši se promjenama poretka (zamjenama sadržaja) objekata u polju A.

Napomena: Iz analize algoritama slijedi da za broj kopiranja records vrijedi $O(n)$ (bar za neke algoritme), tj. nema niti potrebe za dodatnim poljima!

Napomena: Kopiranja records se mogu potpuno izbjeći, ako ne zahtijevamo baš strogo preuređenje niza.

Dovoljno bi bilo naći permutaciju $p \in S_n$

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

(takvih može biti i više, ako ima istih ključeva) koja tako treba preurediti niz,

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

No, pamćenje permutacije p zahtijeva još jedno polje p : array [index] of index.

Osim toga, kod pretraživanja stalno treba konstruirati permutirane indekse - tj. dvostruki indeks (index od indeksa) - što usporava pretragu. Naime, ideja sortiranja je da se traženje obavlja mnogo puta - tako da treba strahovito uštedjeti.

```
{*****************************************************************************

        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 neke tamnije poboljšanja.

– To je, u prosjeku – empirijski, najefikasniji poznati alg. sortiranja (bez stroge dokaza!).

– Osnovna ideja je rekurzivna, vrlo nalik na raspolavljanje odn. binarno pretraživanje:
  - izaberemo neki ključ – tzv. pivotni ključ v_k oko kojeg ćemo raspodijeliti polje na 2 dijela.
  - preuredimo (permutiramo) polje, tako da za neki indeks $k \in \{1, \ldots, n\}$ vrijedi:

    $A[i].key < v_k$ za $i < k$. (može $\leq$, $i \leq k$
    $A[i].key \geq v_k$ za $i > k$. onda $>$, $i > k$)

  - rekurzivno sortiramo 2 manja polja:

    $A[1..k-1]$ i $A[k..n]$.

– Pritome se nadamo da je izabrani pivotni ključ u blizu medijana – polovista, 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: $A[i..j]$ (l, r)

– Drugim riječima, razradujemo (ili projektiramo) potprogram quicksort: (l, r)

```
procedure quicksort (i, j : integer);
```

koji sortira elemente $A[i], \ldots, A[j]$ (l, r) globalnog polja A.

– Oprez – odmah treba uočiti da moramo paziti na to da na ulazu vrijedi $i \leq j$. (l ≤ r)

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

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

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

$$A[\overset{\ell}{i}\,..\,\overset{j-1}{k-1}]$$

prazno, jer je $k = \overset{j=\ell}{l}$ (1)

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

$$A[k\,..\,\overset{(n)}{vj}] = A \quad (\text{jer } k = i)$$

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

- To se lako vjesava: potpolje strogo manjih je korektno definirano, ako i samo ako imamo bar 2 razlicita ključa i x^k nije najmanji (!). (ima bar jedan strogo manji).
- Dalje, kad trazimo x^k - treba provjeriti da li u trenutnom potpolju postoje bar 2 razlicita ključa i x^k ← veći od ta 2.
  U protivnom - svi ključevi su isti, pa ne treba sortirati.
- Ovaj posao obavlja procedura findpivot:

```
                                          ℓ, r
procedure findpivot (i, j : integer;
                     var ok : boolean; {= našao pivot}

                     var pivotindex : index
                       {index pivotnog elementa u polju} );
```

                                          ℓ < r
ujen posao: ako je $i \neq j$ (!) ℓ
  firstkey := A[i].key;
  u tomadu polja $A[\overset{\ell+1}{i+1}\,..\,\overset{r}{j}]$
    trazi prvi ključ razlicit od firstkey.
  ako ga naoteš, pivotni index je index veceg od ta 2 ključa.

Ako smo našli pivotni indeks – treba preurediti polje $A[i..j]$ – particionirati (raspodijeliti) na strogo manje i već ili jednako potpolje oko vrijednosti pivotnog ključa. Osput, treba vratiti i indeks k_0 koji dijeli ta 2 potpolja

$$\underbrace{A[i..k-1]}_{< x_k}, \quad \underbrace{A[k..j]}_{\geq x_k}$$

x_k = pivotni_ključ

Taj posao obavlja procedura partition:

```
                                  l, r
procedure partition (i, j : integer;
                     pivotkey : keytype;
                     var k : integer );
```

Algoritam je standardna dvostrana pretraga – sa svake strane preskačemo sve elemente koji zadovoljavaju uvjet te strane. Kad sa svake strane naletimo pogrešnog – zamijenimo ih, pa oni odlaze na pravu stranu i nastavljamo traženje:

l – dvi indeks trenutnog elem. s lijeve strane
r – –||– ——— ||— s desne strane.

```
l := i; r := j ; { inicijalizacija }
    { velika petlja }
repeat  { može repeat, jer findpivot osigurava l<r }
    { traženje prvog kvivog s lijeve strane ⟶ }
  while A[l].key < pivotkey do l := l+1;
    { l = indeks prvog kvivog, a taj sigurno
      postoji }
    { traženje prvog bivog s desne strane ⟵ }
{if l<r then} while A[r].key ≥ pivotkey do r := r-1;
  if l < r then zamijeni (A[l], A[r])

until l > r;
k := l   (!)   { jer je to baš zadnji kvivi s lijeve,
                 prebačen je na desno i sada je
                 sigurno l = r+1, pa je l dobar
                 indeks desne strane }
```

Potpuni potprogram za Quicksort algoritam:

```
procedure Qsort ( n : zindex;
                  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 );
  { Preureduje 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);
  { Sortira 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 findpivot i partition na $A[i..j]$ i samog sebe, na $A[i..k-1]$ i na $A[k..j]$

Tada je:

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

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

$$T_{findpivot}(i,j) \overset{\in}{=} O(j-i+1) \quad \text{tj. } \leq c_1(j-i+1)$$
$$T_{partition}(i,j) \overset{\in}{=} O(j-i+1) \quad \text{tj. } \leq c_2(-\!\!\!\!-)$$

pa možemo pisati:

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

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

Zbog toga je lakše pisati T_Q kao funkciju duljine polja:

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

Naš polazni zadatak je sortiranje polja duljine n.

Tada je:

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

– Ako imamo sreću, pa je stalno $k \approx \frac{n}{2}$, dobivamo rekurziju oblika $\lceil \frac{n}{2} \rceil$

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

s rješenjem: $T_Q(n) = O(n \log n)$.

- U najgorem slučaju je $k = 2$ ili n, tj. potpolja su duljine 1 i $n-1$.
  Očito je $T_Q(1) \leq c''$, pa izlazi:

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

  s rješenjem: $T_Q(n) = O(n^2)$.

- Lako se vidi da je i na donjoj ogradi

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

jer svaki poziv procedure (bilo koje) ponešto traje. Ovo vrijedi ako su svi ključevi međusobno različiti (inače findpivot može skratiti vrijeme na $\Theta(n)$).

- Najgora složenost se može dostići. Primjer je naopako sortirano polje (silazno poredano). Tada je pivotni ključ uvijek <u>najveći</u> element potpolja, tj. desno potpolje je uvijek duljine 1.

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

- Zamjene se javljaju 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: $N_{zamjena}(n) = O(n^2)$.

- Srećom, u prosjeku, dobivamo mnogo bolju ocjenu.

Prosječna vremenska složenost Quicksorta

Za prosječnu složenost moramo povesti pretpostavke o distribuciji ulaza, i definirati prosj. složenost.

Polazne pretpostavke:

(*) – svi polazni poretci (redoslijedi – permutacije) su jednako vjerojatni,
– nema jednakih ključeva
(ovo je samo zbog pojednostavljenja)
Naime, isti ključevi samo uhrzavaju algoritam jer se taj dio, zaporavo, ne sortira.

To znači da imamo n! jednako vjerojatnih poredaka na ulazu. Prosječno vrijeme gledamo kao srednju vrijednost po svim polaznim poretcima. (perm.)

– Za pojednostavljenje analize, uvodimo još jednu pretpostavku:
– u trenutku kad pozivamo quicksort(i,j) na A[i..j], svi poreci za A[i], ..., A[j] su, također, jednako vjerojatni.
(Ovo je mnogo strože! Ranija pretp. je ovo isto, ali samo za poziv quicksort(1,n)).

Opravdanje: (umjesto strogog dokaza)
Prije svog poziva, svi ranији pivotni ključevi su napravili particiju – rez u tom dijelu polja A[i..j].
Tj. svi oni elementi su manji ili strogo manji od v ili su svi veći ili jednaki v.
(To je ⇐ kao da ajeli svaj blok interpretiramo kao 1 element – bez pojedinačnog razlikovanja)

Precizније: može se (pedantnom analizom) dokazati da bilo koji pivot ima veću šansu da završi blizu desnog ruba desnog potpolja (onih 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 $T(1) = c_1$ (ili $\leq c_1$) 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 uvlačimo u faktor c_2, tj. umjesto $c_2 n + c$, pišemo $c_2 n$, jer $c > 0$).

Ova ocjena 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 ($i \in \{1, \ldots, n-1\}$ – 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

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

Vidimo, da zbog pretp. da su svi ključevi 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 nalaženje 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 prvom mjestu, a jedan od i elemenata manjih od njega je na drugom mjestu.

Zbog pretp. o jednakoj vjerojatnosti svih ulaznih poredaka, vjeroj. da je bilo koji odredeni element (na pr. i+1-i po redu veličini) na prvom 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 opaj 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 ljevo potpolje duljine i.

- Rekurzija 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 koristimo relaciju:

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

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

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

$$T(n) \leqslant \underbrace{\frac{2}{n-1}}_{\hookrightarrow \text{ovdje može i } n,\ \text{ali ne treba!}} \sum_{i=1}^{n-1} T(i) + c_2 n$$

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

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

- Ova rekurzija se može i direktno rješavati (sami).
- Nama je dovoljno dokazati da $\exists c > 0$ konst. takva da je
$$T(n) \leqslant c \cdot n \cdot \log n, \quad \forall n \geqslant 2$$
što dokazujemo matematičkom indukcijom.

- Baza je za $n = 2$. Tada je
$$T(2) \leqslant 2 \cdot T(1) + 2c_2 = (\text{tj.} \leqslant)\ 2c_1 + 2c_2$$
pa iz $T(2) \leqslant c \cdot 2 \log 2$ slijedi
$$c_1 + c_2 \leqslant c \cdot \log 2$$
tj.
$$\underbrace{c \geqslant \frac{c_1 + c_2}{\log 2}}.$$

Pretpostavimo da je c tako odabran i da vrijedi
$$T(i) \leqslant c \cdot i \cdot \log i, \quad i = 1, \dots, n-1$$

Iz rekurzije je:
$$T(n) \leqslant \frac{2c}{n-1} \sum_{i=1}^{n-1} i \log i + c_2 n$$

Ova suma rastavljamo u 2 podjednaka dijela koje onda ocjenjujemo:

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

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

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

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

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

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

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

$\frac{n(n-1)}{2}$

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

Ako stavimo $c \geq 4c_2$ suma 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 := l ;
  j := r + 1 ;
{
  pivot_index := l ;
}
  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 := l ;
  j := r ;
  pivot_index := ( l + 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 }

{*****************************************************************************}
```