Др Ђура Паунић

ПРИМЕР ПОБОЉШАЊА АЛГОРИТМА СОРТИРАЊА

Када се скуп састоји од елемената за које постоји дефинисано линеарно уређење, тада је једна од врло важних и честих операција у програмирању премештање елемената тако да се добије низ уређен по величини, тј. да уређење елемената одговара уређењу индекса низа.

Дефиниција. Операција премештања елемената низа тако да се после њеног извођења добије низ у коме су елементи уређени по величини се назива сортирање низа.

Нека је скуп имплементиран коришћењем низа (`array`), дакле елементи скупа су смештени у низ дужине `maxniz`. О упоредивом типу по коме се елементи скупа упоређују, се ништа не претпоставља тј. за два елемента можемо да утврдимо само који је већи или да ли су једнаки. Користићемо максимално поједностављену репрезентацију елемената скупа тј. уместо да су елементи скупа слогови који имају поље `kljuc` по коме се сортирају, претпоставићемо да су сами елементи низа упоредивог типа. Користићемо тип `integer` да се за упоређивање елемената користе релације $<$ и $\leqslant$ из програмског језика, иако за целе бројеве постоје и друкчији поступци сортирања у којима се користи структуру целог броја, тј. да се он састоји од цифара.

```
const
        maxniz = 100;  (* neka pogodna konstanta *)
type
        element = integer;
        niz = array[1 ..  maxniz] of element;
```

Ако је поредак линеаран тада је скуп могуће уредити или растуће или опадајуће. Како су ова два начина сортирања еквивалентни претпоставимо у даљем излагању да се скуп увек уређује у растућем поретку (или неопадајуће уколико се допушта да у низу постоје једнаки елементи).

Дакле, ако се уређење у скупу означи са $\leqslant$ тада се после сортирања низа a добија низ за који важи

$$a[1] \leqslant a[2] \leqslant \ldots \leqslant a[maxniz].$$

1. Елементарни поступци сортирања низова

Низ може да се сортира на више начина. Начини сортирања се деле на тзв. елементарне поступке сортирања, оне у којима се директно користе основне идеје за сортирање, и сложене поступке сортирања у којима се побољшавају неки недостаци елементарних поступака.

Елементарни поступци сортирања састоје се у томе да се у сваком кораку величина сортираног дела низа повећава за један, а величина несортираног дела за један смањује. Ради једноставнијег објашњавања алгоритма сортирања претпоставићемо да се на почетку низа налази несортирани део, а на крају сортирани. Ако се жели да сортирани део буде на почетку низа тада треба алгоритме модификовати тако да се неке петље изводе у супротном смеру.

У основи је могуће сортирати низ користећи варијације следећа три поступка:

- Сортирање уметањем (енгл. insertion sort), је поступак за сортирање у коме се у сваком кораку последњи елемент из несортираног дела низа пребаци у помоћну променљиву и затим се упоређује са елементима у сортираном делу. Док су елементи у сортираном делу мањи од њега они се померају у лево, а када се у сортираном делу нађе место на коме треба да се налази тај елемент он се убацује, умеће, на нађено место.
- Сортирање изабирањем (енгл. selection sort), је поступак за сортирање у коме се у сваком кораку у несортираном делу низа налази највећи елемент и он постави на крај несортираног дела. На тај начин се добија да су у сваком кораку сви елементи несортираног дела мањи од свих елемената сортираног дела.
- Сортирање разменом (енгл. exchange sort), је поступак за сортирање у коме се у сваком кораку пролази кроз несортирани део и два суседна елемента размене место ако „стоје погрешно“. После једног таквог пролаза, када се полази од најмањег ка највећем индексу несортираног дела низа, на крају несортираног дела низа ће се наћи највећи елемент у несортираном делу. И при овом поступку су у сваком кораку сви елементи несортираног дела мањи од свих елемената у сортираном делу низа.

Дакле, све елементарне методе сортирања функционишу тако, што се низ подели на два дела, сортирани део и несортирани део, а у сваком алгоритамском кораку се сортирани део повећава за један елемент.

Објаснимо детаљније сортирање изабирањем и његово побољшање.

2. Сортирање изабирањем

За сортирање изабирањем се користи чињеница да ако су сви елементи у сортираном делу низа већи од свих елемената у несортираном делу низа и ако се у несортираном делу нађе највећи елемент и постави на крај несортираног дела да се тада сортирани део повећа за један, а и даље су сви елементи у несортираном делу мањи од свих елемената у сортираном делу низа.

На почетку је сортирани део низа празан и најпре се нађе највећи елемент у целом низу, па се пребаци на крај низа. Затим се у несортираном делу низа, од првог елемента па до претпоследњег, опет нађе највећи и он пребаци на претпоследње место итд. У последњем кораку се први и други елемент упоређују и већи стави на друго место. Овај поступак је реализован у следећој процедури.

```
procedure SelectionSort(var a :  niz);
var
    i, j, m :  integer;
    temp :  element;
begin
    for i := maxniz downto 2 do
        begin
            (*  Neka je prvi element najveci  *)
            m := 1;
            temp := a[1];
            (*  Proveriti da li je on stvarno najveci
                  uporedjivanjem sa ostatkom niza      *)
            for j := 2 to i do
                if a[j] > temp then
                    begin
                        m := j;
                        temp := a[j]
                    end;
            (* Prebaciti najveci element
                 na i-to mesto             *)
            a[m] := a[i];
            a[i] := temp
        end
end; (* SelectionSort *)
```

Када постоји више алгоритама за решење једног проблема тада треба утврдити који је од њих бољи, тј. у овом случају који поступак брже сортира елементе. При том треба бројати укупан број корака, који очигледно зависи од величине низа, а у детаљнијој анализи укупан број упоређивања и укупан број премештања.

С друге стране брзина сортирања ће зависити и од почетног распореда елемената у низу који се сортира. У анализи сортирања се претпоставља да се сортира низ од различитих елемената, јер ако постоје елементи који се понављају, тада је обично потребно мање корака, а прецизна анализа је врло компликована, јер зависи и од распореда елемента који се понављају.

Најчешће се анализирају три случаја који зависе од распореда елемената. Посматра се број операција при најповољнијем распореду елемената за алгоритам, при најнеповољнијем распореду и средњи број операција. Под средњим

бројем операција се подразумева број операција у случају када се претпостави да је сваки распоред елемената једнако вероватан.

Израчунајмо број упоређивања, C (C_{min} минималан, C_s средњи и C_{max} максималан), и број премештања елемената низа, M (M_{min} минималан, M_s средњи и M_{max} максималан), за сортирање изабирањем.

Добија се да при сортирању изабирањем број упоређивања не зависи од распореда елемената, тј. увек је исти:

$$\begin{aligned} C_{min} &= C_s = C_{max} = (n-1) + (n-2) + \cdots + 2 + 1 \\ &= \frac{n(n-1)}{2} = \frac{1}{2}n^2 + O(n), \end{aligned}$$

док за број премештања важи да имамо $n-1$ корак, а у сваком кораку имамо:

$$\begin{aligned} M_{min} &= 3 + 3 + \cdots + 3 = 3(n-1) = 3n + O(1), \\ M_{max} &= (3 + n - 1) + (3 + n - 2) + \cdots + (3 + 1) \\ &= 3(n-1) + \frac{n(n-1)}{2} = \frac{n^2 + 5n - 6}{2} = \frac{1}{2}n^2 + O(n), \end{aligned}$$

Израчунати M_s је мало сложеније. Објаснимо на првом кораку. Вероватноћа да `temp` треба да се замени са `a[j]` за `j = 2` је $1/n$, за `j = 3` је $1/(n-1)$, итд. за `j = maxniz` је $1/2$. У следећим корацима је потпуно аналогно резоновање, само је низ на који се примењује краћи, па се добија:

$$\begin{aligned} M_s &= \left(\frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{2}\right) + \left(\frac{1}{n-1} + \cdots + \frac{1}{2}\right) + \cdots + \frac{1}{2} + 3(n-1) \\ &= \frac{n-1}{2} + \frac{n-2}{3} + \frac{n-3}{4} + \cdots + \frac{n-(n-1)}{n} + 3(n-1) \\ &= \frac{n+1-2}{2} + \frac{n+1-3}{3} + \cdots + \frac{n+1-n}{n} + 3(n-1) \\ &= (n+1)\left(\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n}\right) + 2(n-1) \\ &= (n+1)h_n + n - 3 \approx n\ln n + (1+\gamma)n + \ln n + \gamma - 3 \\ &= n\ln n + O(n). \end{aligned}$$

У извођењу је искоришћена оцена за h_n, хармонијске бројеве,

$$h_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \ln n + \gamma + O(\frac{1}{n}),$$

где је γ Ојлерова константа ($\gamma = 0.5772156649\ldots$).

Сортирање изабирањем може да се побољша тиме да се користи бржи поступак за налажење највећег елемента у низу од секвенцијалног претраживања. За то се користи приоритетна листа.

3. Приоритетна листа – Апстрактни тип података

Приоритетна листа (енгл. heap, priority queue) се састоји од скупа упоредивих елемената, при чему су операције које нас интересују приступ највећем елементу, избацивање елемента са највећим приоритетом и додавање елемента произвољног приоритета. Дакле, треба одабрати имплементацију у којој се ове три операције што ефикасније изводе.

У даљем излагању ћемо претпоставити да је приоритет цео број. При том су могућа два, очигледно еквивалентна приступа, да већи број буде и већи приоритет или обрнуто да је већи број мањи приоритет. Претпостављаће се да већи број означава већи приоритет. Да би скуп имао структуру приоритетне листе треба да су дефинисане следеће операције:

1. `MakeNull(S)`. Направити празну приоритетну листу.
2. `Insert(x, S)`. Убацити елемент x у приоритетну листу S, тако да се структура приоритетне листе сачува.
3. `DeleteMax(S, ok)`. Избацити елемент са највећим приоритетом из листе S тако да се структура приоритетне листе сачува. Променљива ok је `true`, ако је операција успешно изведена, а иначе `false`.
4. `MaxHeap(x, S, ok)`. У елемент x копирати елемент приоритетне листе S који има највећи приоритет, наравно уколико постоји.
5. `MakeHeap(A, S)`. Од елемената произвољног низа A направити приоритетну листу S.

Често се јавља потреба и за још неким операцијама са приоритетним листама, на пример промена приоритета елемента, спајање две приоритетне листе у једну и сл, али их овде нећемо посматрати.

Приоритетна листа може врло једноставно да се реализује користећи бинарно стабло специјалног облика, при чему је у корену елемент са највећим приоритетом. Ради једноставности у даљем излагању ће се претпостављати да су елементи приоритетне листе цели бројеви и да већи број има већи приоритет. У општем случају је елемент скупа слоговног типа који има поље `prioritet` по коме се упоређују приоритети елемената.

Пример 1. Од елемената 4, 6, 7, 8, 9, 10, 12, 13, 14, 15, 17 и 18 може да се сложи бинарно дрво приказано на слици а) које има особине приоритетне листе.

а) б)

Сл. 1. Приказ приоритетне листе

Бинарно стабло којим се представља приоритетна листа има следеће особине:

1. за сваки чвор у стаблу важи да су синови мањи од оца,
2. сви нивои су попуњени осим последњег,
3. у последњем нивоу су сви чворови, почев од левог, редом попуњени докле год има елемената.

Овај облик бинарног стабла се лако може запамтити сликом б) која на њега подсећа.

Овако описана приоритетна листа се ефикасно реализује помоћу низа. Наиме, могуће је сместити бинарно стабло у низ по нивоима редом. На првом месту у низу је корен, на местима 2 и 3 је други ниво, на местима 4, 5, 6 и 7 трећи итд. Тада се добија да ако је неки чвор смештен на позицију k, његови синови су на позицијама $2k$ и $2k+1$, а чвор чији је индекс n има за оца чвор чији је индекс n `div 2`. При том је суштински битно да индекс низа почиње од један. Да би се потпуно одредило бинарно стабло још једино треба упамтити позицију последњег елемента.

Пример 2. Сместимо у низ стабло из претходног примера. Треба уочити да се елементи у низ ређају по нивоима. Испод бројева у низу су њихови индекси.

```
elementi  18, 17, 14, 15, 13,  9,  8, 10,  4,  7, 12,  6.
indeksi    1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12.
```

```
const
    maxniz = 100;
type
    element = integer;
    niz = array[1 ..  maxniz] of element;
    plista = record
                  elementi :  niz;
                  zadnji :  integer
             end;
```

За иницијализацију листе довољно је поставити индекс задњег елемента на вредност мању од најмањег индекса низа, тј. на нулу, јер се при додавању елемената индекс задњег елемента повећава.

```
procedure MakeNull(var s :  plista);
begin
      s.zadnji := 0
end; (* MakeNull *)
```

Убацивање елемента је једноставно. Реализује се на следећи начин:

- уколико у низу има слободног места, нови елемент се додаје на крај низа и повећа се граница последњег елемента,

- упоређује се приоритет новог елемента са приоритетом његовог оца. Уколико је приоритет новог елемента већи од приоритета оца тада та два елемента замене места и настави се овај поступак упоређивања приоритета, док се за нови елемент не нађе одговарајуће место у приоритетној листи, тј. док је приоритет новоубаченог елемента већи од елемента оца.

Из образложења поступка убацивања елемента следи, да се убацивање новог елемента изводи за $O(\log n)$ операција, јер се индекс оца елемента x добија тако, што се преполови индекс елемента x.

Поступак убацивања може да се лако реализује и рекурзивно и итеративно. Изложимо итеративну верзију процедуре. Ако у листи има места за нови елемент тада се повећа `a.zadnji` за један и та вредност се додели променљивој `mesto`. Затим се упоређују приоритети новог елемента и елемента са индексом `otac := mesto div 2`. Док је приоритет новог елемента, `x`, већи од приоритета елемента у листи чији је индекс `otac` помера се елемент са позиције `otac` на позицију `mesto` и израчунавају нови индекси (`mesto := otac; otac := mesto div 2`), а када се дође до елемента у листи са већим приоритетом од `x` или до корена стабла нови елемент се убаци у листу.

```
procedure Insert(x :  element;
        var s :  plista;
        var ok :  boolean);
var
  mesto, otac :  integer;
  gotovo :  boolean;
begin
  with s do
    if zadnji = maxniz then
      ok := false
    else
      begin
        ok := true;
        zadnji := zadnji + 1;
        if zadnji = 1 then
          elementi[zadnji] := x
        else
          begin
            mesto := zadnji;
            gotovo := false;
            repeat
              otac := mesto div 2;
              if x > elementi[otac] then
                begin
```

```
                elementi[mesto] := elementi[otac];
                mesto := otac
              end
            else
              gotovo := true
          until (mesto = 1) or gotovo;
          elementi[mesto] := x
        end
    end
END Insert;
```

У Турбо Паскалу и другим програмским језицима, у којима се не израчунава вредност другог израза у конјункцији када први има вредност `false`, могуће је поједноставити ову процедуру да се не користи додатна логичка променљива `gotovo`, али је нешто теже схватити да се елемент убацује на право место.

```
procedure InsertTP(x :  element;
        var s :  plista;
        var ok :  boolean);
var
  mesto, otac :  integer;
begin
  with s do
    if zadnji = maxniz then
      ok := false
    else
      begin
        ok := true;
        zadnji := zadnji + 1;
        mesto := zadnji;
        if zadnji > 1 then
          begin
            otac := mesto div 2;
            while (mesto > 1) and (x > elementi[otac]) do
              begin
                elementi[mesto] := elementi[otac];
                mesto := otac;
                otac := mesto div 2
              end
          end;
        elementi[mesto] := x
      end
end; (* InsertTP *)
```

Процедура за избацивање елемента са највећим приоритетом из приоритетне листе, `DeleteMax`, такође је једноставна. Да би се то постигло на прво место у низу се копира последњи елемент из листе и дужина листе се скрати за један. Тиме је избачен елемент са највећим приоритетом и листа је скраћена, али је покварена структура приоритетне листе, осим ако је листа имала 1 или 2 елемента. Да би се опет успоставила структура приоритетне листе потребно је мало посла, јер само први елемент у листи није на свом месту. Поправљање структуре се постиже упоређивањем приоритета елемента у корену са приоритетима његових синова. Елемент у корену се замењује са оним од својих синова који има највећи приоритет. Овај поступак замене елемента са његовим сином који има највећи приоритет се наставља докле год постоји бар један син који има већи приоритет од оца или се дође до краја листе.

```
procedure DeleteMax(var s :  plista;
        var ok :  boolean);
var
  mesto, sledeci :  integer;
  nastavi :  boolean;
  temp :  element;
begin
  with s do
    if zadnji = 0 then
      ok := false
    else
      begin
        ok := true;
        if zadnji = 1 then
          zadnji := 0
        else
          begin
            elementi[1] := elementi[zadnji];
            zadnji := zadnji - 1;
            if zadnji > 1 then
              begin
                mesto := 1;
                sledeci := 2 * mesto;
                temp := elementi[mesto];
                nastavi := true;
                while (sledeci <= zadnji) and nastavi do
                  begin
                    if sledeci < zadnji then
                      if elementi[sledeci+1] >
```

```
                        elementi[sledeci] then
                        sledeci := sledeci + 1;
                    if elementi[sledeci] > temp then
                      begin
                        elementi[mesto] := elementi[sledeci];
                        mesto := sledeci;
                        sledeci := 2 * mesto
                      end
                    else
                      nastavi := false
                  end;
                elementi[mesto] := temp
              end
          end
      end
end; (* DeleteMax *)
```

Процедура `DeleteMax` такође може да се једноставно реализује и рекурзивно.

Из поступка је јасно да се и операција избацивања елемента са највећим приоритетом изводи такође за $O(\log n)$ операција, јер је индекс сина $2k$ или $2k+1$, ако је индекс оца k, тј. у сваком кораку се индекс бар удвостручује па се за $O(\log n)$ корака долази до краја листе.

Налажење елемента са највећим приоритетом, процедура `MaxHeap`, је врло једноставна и он се налази за $O(1)$ корака.

```
procedure MaxHeap(var x :  element;
        var s :  plista;
        var ok :  boolean);
begin
  with s do
    if zadnji = 0 then
      ok := false
    else
      begin
        ok := true;
        x := elementi[1]
      end
end; (* MaxHeap *)
```

Процедура `MakeHeap` се реализује тако што се постепено све већи и већи део низа претвара у приоритетну листу користећи сличан поступак као у процедури `DeleteMax`. Упоређују се елементи са својим синовима и смештају се

на одговарајуће место почев од средине низа (`kraj div 2`) ка почетку, јер је тада потребно испитати само `kraj div 2` елемената. Реализујмо сваки корак овог поступка помоћном рекурзивном процедуром `rheap`, која је слична процедури `DeleteMax`, једино у њој корен стабла не почиње са 1 него од произвољног елемента у низу. Процедура `MakeHeap` има и параметар `kraj`, индекс последњег елемента у низу.

```
procedure MakeHeap(a :  niz;
        kraj :  integer;
        var s :  plista;
        var ok :  boolean);
var
  i :  integer;
  procedure rheap(levi :  integer);
  var
    sledeci :  integer;
    temp :  element;
  begin
    sledeci := 2 * levi; (* indeks prvog sina *)
    if sledeci <= kraj then
      with s do
        begin
          (* naci indeks najveceg sina *)
          if sledeci < kraj then
            if elementi[sledeci+1] > elementi[sledeci] then
              sledeci := sledeci + 1;
            (* zameniti ako treba *)
            if elementi[sledeci] > elementi[levi] then
              begin
                temp := elementi[levi];
                elementi[levi] := elementi[sledeci];
                elementi[sledeci] := temp;
                rheap(sledeci)
              end
        end
  end; (* rheap *)
  begin
    if kraj <= maxlista then
      begin
        ok := true;
```

```
        with s do
          begin
            elementi := a;
            zadnji := kraj
          end;
        if kraj > 1 then
          for i := kraj div 2 downto 1 do
            rheap(i)
      end
    else
      ok := false
  end; (* MakeHeap *)
```

За конструкцију приоритетне листе треба у листу убацити n елемената, а за свако убацивање треба $O(\log n)$ корака, на први поглед укупно $O(n \log n)$ корака. Међутим, пажљивијим бројањем се добија да се трансформација низа у приоритетну листу изводи за $O(n)$ корака. Наиме, за елементе са индексима из интервала $(n/4+1, n/2)$ има највише једно премештање, за елементе са индексима у интервалу $(n/8+1, n/4)$ највише два премештања итд. Коначно се добија да је за конструкцију приоритетне листе од низа укупан број премештања

$$\frac{n}{4} + \frac{2n}{8} + \frac{3n}{16} + \cdots \approx \frac{n}{4}\left(1 + 2\frac{1}{2} + 3\frac{1}{2^2} + \cdots\right) = \frac{n}{4}\frac{1}{\left(1-\frac{1}{2}\right)^2} = n.$$

Приоритетна листа се може реализовати и тако да сваки елемент у листи уместо два сина има d синова, $d > 2$, дакле за реализацију се не користи бинарно стабло него стабло арности d. У том случају се скраћује висина стабла, па самим тим и дужина пута при убацивању или избацивању, али се компликује избор сина са највећим приоритетом, тако да се обично користе приоритетне листе са $d = 3$ или 4, јер се иначе губи сувише времена на тражење највећег сина.

4. Heapsort

Следећа метода сортирања је усавршење сортирања изабирањем. У горе описаном поступку сортирања изабирања несортирани део низа није имао никакву структуру и највећи елемент у њему се тражио секвенцијално. Међутим, налажење највећег елемента је врло једноставно (за $O(1)$ корака) ако несортирани део има структуру приоритетне листе, јер тада највећи елемент има индекс 1.

Нека несортирани део низа, на почетку цео низ, има структуру приоритетне листе и то тако да већи приоритет има већи елемент. Тада је елемент са највећим приоритетом на првој позицији у низу. Затим, замене место он и последњи елемент низа, приоритетна листа се скрати за један и поново успостави структура приоритетне листе. Тиме се добија уређен низ од једног елемента на крају низа, сви елементи у уређеном делу су већи од свих елемената у неуређеном делу и неуређени део, приоритетна листа, је за један краћи. Ако се овај поступак

настави тада је у сваком кораку у делу низа са индексима од 1 до k приоритетна листа, при чему је први елемент највећи у том интервалу, и сортиран део низа у интервалу са индексима од $k+1$ па до краја низа. При том су сви елементи у сортираном делу већи од свих елемената на почетку низа. Када се први елемент замени са k-тим и поново успостави структура приоритетне листе на интервалу од 1 до $k-1$ добија се да је сортирани део низа повећан за 1 и да су сви елементи у сортираном делу већи од свих елемената у приоритетној листи. Из описа је јасно да се може направити и итеративан и рекурзиван поступак. Итеративна верзија је:

```
procedure HeapSort(var a :  niz);
var
  levi, desni :  integer;
  temp :  element;
  procedure Heap(levi, desni :  integer);
  var
    i, j :  cardinal;
    gotovo :  boolean;
  begin
    i := levi;
    j := 2*i;
    temp := a[i];
    gotovo := false;
    while (j <= desni) and not gotovo do
      begin
        (* izaberimo veceg od sinova ako postoji *)
        if j < desni then
          if a[j] < a[j+1] then
            j := j + 1;
          if a[j] <= temp then
            gotovo := true
          else
            begin
              a[i] := a[j];
              i := j;
              j := 2*i
            end
      end;
    a[i] := temp
  end; (* Heap *)
begin
```

```
    (* Najpre napraviti prioritenu listu *)
    levi := (maxniz div 2) + 1;
    desni := maxniz;
    while levi > 1 do
      begin
        levi := levi - 1;
        Heap(levi, desni)
      end;
    (* Redom prebacivati najvece elemente
      na kraj prioritetne liste *)
    while desni > 1 do
      begin
        temp := a[1];
        a[1] := a[desni];
        a[desni] := temp;
        desni := desni - 1;
        Heap(1, desni)
      end
  end; (* HeapSort *)
```

Уместо `while`-петљи могу да се користе `for`-петље, а процедура `Heap` у `HeapSort`-у може да се реализује и рекурзивно

Анализа хипсорта је врло једноставна и не зависи од полазног распореда елемената у низу. Најпре се у низу направи приоритетна листа, за то је потребно $O(n)$ операција. Затим се n пута највећи елемент премешта на крај листе и успоставља структура приоритетне листе. За ово је потребно $O(n \log n)$ операција без обзира на распоред елемената. Дакле, све у свему $O(n \log n)$ операција.