Ана Капларевиќ-Малишиќ

О АЛГОРИТМИМА – СОРТИРАЊЕ

Прва група алгоритама о којима ћемо говорити јесу, алгоритми који имају задатак да елементе једне листе† уреде према одређеном критеријуму. Најчешће коришћена уређења су нумеричко и лексикографско. У пракси, ефикасни алгоритми за сортирање, се углавном користе за припрему улазних података за друге алгоритме којима су сортирани улази потребни да би исправно радили, као што су алгоритми претраге или спајања више уређених листа у већу.

Класификација алгоритама за сортирање се може обавити према:

- комплексности израчунавања (у најгорем, просечном и најбољем случају) у зависности од величине улаза
- величини додатног меморијског простора који им је потребан да би задатак обавили, опет у зависности од величине улаза
- стабилности.

Стабилност алгоритама има смисла дефинисати ако сортирају листе сложених стуктура (нпр. ако су елементи листе слогови који садрже више података као што су: редни број, име, презиме, одељење). Стабилан алгоритам је онај који чува релативни (међусобни) поредак елемената листе, ако им је кључ, тј. поље по коме се врши уређивање, исти. То значи да ако елементи листе R и S имају једнаке вредности кључува, а у оригиналној (несортираној) листи имају такав распоред да је R испред S, онда такав распоред мора да важи и у сортираној листи.

Списак познатих стратегија за сортирање је прилично дуг. Представићемо вам неке од њих.

Selection sort

Овај начин сортирања је, вероватно, најприроднији и интутивно јасан. Идеја која се њиме реализује јесте да се за свако место одреди који елемент би требао на њему да се нађе. Наиме, прво место у листи је намењено минималном елементу листе, друго другом по величини итд. Алгоритам сваку позицију појединачно, почевши од прве, намешта тако што пореди елемент који се тренутно налази на тој позицији са свим елементима који се налазе десно од њега. У случају да се десно нађе мањи, елементи размењују места. Дакле, у i-том пролазу, сви елементи са индексом мањим од i су већ на одговарајућим позицијама (сортирани). Из тог разлога се i-ти пореди само са елементима десно од њега. Побољшана верзија овог алгоритма би се могла добити одређивањем позиције минималног међу елементима од i-те до n-те позиције, а затим вршењем само једне размене између i-тог и пронађеног елемента.

Овакав алгоритам увек врши исти број поређења, без обзира на распоред елемената у улазној листи, те, у сваком случају, његова брзина зависи само од величине, али не и од квалитета улаза и припада класи $O(n^2)$. Ако посматрамо број размена, у најгорем случају је $O(n)$ (у побољшаној верзији).

† Овде се не мисли на листу као динамичку структуру података, већ на било какву листу елемената, тј. скуп у коме сваки елемент има своју позицију.

Insertion sort (сортирање уметањем)

Као што му име каже, овај алгоритам сортира тако што умеће сваки елемент листе на одговарајуће место у сортираној листи. Најједноставнија имплементација ове идеје захтева две листе - поред улазне листе користи се посебна листа за смештање уређених елемената. Наравно, таква реализација није рационална. Верзија која не захтева додатни меморијски простор би имала мало другачији ток. Наиме, ако i-та ($i = 1, \ldots, n-1$) итерација подразумева да је првих i елемената већ сортирано, тада се $i+1$-вом елементу, нека је то x, „тражи“ место међу претходним члановима. Ако је тражена позиција k-та ($k \leqslant i$) елементи низа од k-тог до последњег се померају удесно, док се x смешта на k-ту позицију .

Комплексност овог алгоритма је у најгорем и просечном случају $O(n^2)$, ако говоримо о броју поређења. Број размена је највише $O(n)$. Додајмо још да спада у стабилне алгоритме.

Bubble sort

Bubble sort је стабилан алгоритам и функционише тако што пореди свака два суседна елемента мењајући им места ако нису у добром распореду. Прецизније, он има следећи ток:

- Врши се поређење између свака два суседна елемента, почеши од почетка листе, и ако је потребно елементима се размењују вредности. Након поређења последњег пара, на крају листе ће се наћи највећи елемент.
- Понавља се поступак поређења парова, при чему се за сваку серију поређења изоставља по један елемент здесна више у односу на претходни пролаз, све док се не позиционирају сви елементи.

Мерено бројем поређења комплексност овог алгоритма је $O(n^2)$. Што се броја размена тиче, у најгорем случају он је $O(n^2)$.

Counting sort (сортирање пребројавањем)

Овај, такође стабилан, алгоритам се користи за сортирање листа код којих је кључ по коме се сортира природан, односно цео број (мада се идеја може искористити за све врсте листа код којих је кључ из неког коначног опсега). За његов рад је потребно одређивање максималне вредности кључа (ако су природни бројеви питању). Идеја је да се за формирање излазне листе користи помоћна листа, чији су индекси вредности у интервалу од 0 до максималне вредности кључа. Та листа има задатак да преброји појављивања свих бројева из скупа „могућих“ вредности, и добија се једним пролазом кроз улазну листу. На основу помоћне листе алгоритам реконструише сортирану излазну листу.

Комплексност овог алгоритма је $O(n+k)$, где је k максимална вредност кључа. Очигледно, k би требао да буде осетно мањи у односу на n, да не би имао битног утицаја на брзину рада.

Merge sort (John von Neumann, 1945)

Овај алгоритам спада у оне који проблем решавају по принципу „завади, па владај“. Припада стабилним алгоритмима. Његов концепт је следећи: дели несортирану листу на две подлисте приближно исте дужине, сортира обе и спаја их у једну сортирану. Резултат се смешта у нову листу. Код би могао овако да изгледа

```
procedure sort(var a:niz; begin,end:integer);
var mid:integer;
begin
if (begin<>end) and (begin<>end-1) then begin
   mid:=trunc((begin+end)/2);
   sort(a,begin,mid);sort(a,mid,end);
   merge(a, begin, mid, end);
   end;
end;
```

Очигледно, овај алгоритам ради рекурзивно. Поставља се питање: где је ту поређење? Оно се крије у процедури merge. Почиње се од спајања једночланих листа и заменом њихових места ако је елемент из прве листе већи од елемента из друге листе. Након тога се две двочлане листе спајају у једну четворолану итд. Спајање две вишечлане сортиране листа се може извести једним пролазом кроз обе листе, а сам поступак није компликован.

Best case и worst case улази дају брзину $O(n \log n)$, ако је мерило комплексности број поређења. Што се броја премештања (размена) тиче, у најгорем случају комплексност износи $O(n \log n)$, а у најбољем $\theta(\frac{1}{2} n \log n)$. Стандардна имплементација merge sort-а не подразумева сортирање низа на лицу места, већ је за то потребан додатни меморијски простор величине улаза у који би се смештао сортирани излаз. Избегавање ове ставке је могуће, али би алгоритам био изузетно тежак за имплементацију.

Radix sort (Harold H. Seward, MIT, 1954)

Ову врсту алгоритма могуће је користити ако је кључ, тј. поље по коме се врши сортитање листе, број или низ карктера (или стринг). Први корак је дефинисање низа могућих вредности појединачних карактера у стрингу, односно цифара у броју. Ради једноставности објашњења узећемо да је кључ цео број у декадном систему. Дакле, вредности цифара су $0, 1, 2, \ldots, 9$. Сортирање има следећи ток:

- у првом пролазу врши се сортирање елемената листе према цифри јединица, али тако да међусобни распоред елемената из исте групе (оних који имају исту цифру јединица) не мења у односу на полазну листу; као резултат добијамо 10 листа (виртуелно или стварно, свеједно, замислите их као 10 кутија) које се назад спајају у једну
- у следећем кораку се врши сортирање претходно добијене листе према цифрама десетица, опет чувајући распоред елемената са истим цифрама десетица; као резултат добило би се следеће, број који је био, нпр., у кутији 8 (цифра јединица је 8) због цифре десетица која је, нпр., 1 прелази у кутију 1 (и то на крај листе 1)
- понављање корака и за све остале цифре (стотина, хиљада итд.)

Обзиром на то како је осмишљен, очигледно је да и овај алгоритам спада у стабилне.

Пример: Посматрајмо низ 36 9 0 25 1 49 64 16 81 4. Први пролаз даје следећи распоред у кутијама

Кутија	0	1	2	3	4	5	6	7	8	9
Садржај	0	1			64	25	36			9
		81			4		16			49

тј. добијамо следећи распоред у листи 0 1 81 64 4 25 36 16 9 49;

Након другог пролаза (што је уједно и последњи пролаз јер су бројеви максимално двоцифрени) састав кутија је следећи

Кутија	0	1	2	3	4	5	6	7	8	9
Садржај	0	16	25	36	49	–	64	–	81	–
	1									
	4									
	9									

тј. добијамо сортирану листу 0 1 4 9 16 25 36 49 64 81.

Напоменимо да чак није неопходно користити исту основу у свакој фази сортирања, јер делови кључа (ако кључ није једноставан као што је број, већ на пример слог са пољима дан, месец, година, који имају различите опсеге). Што се комплексности овог алгоритма тиче, уопштено гледајући алгоритам она је $O(n \cdot s)$ где је s дужина кључа. Ако се сортирање примењује на низове целих бројева, где је дужина кључа релативно мала, у сваком случају, ограничена, комплексност се своди на $O(n)$. Ако постоји услов да се кључеви у листи од n елемената не понављају, онда се ситуација мења, јер са растом броја елемената мора да расте и величина кључа. Наиме, ако имамо n елемената и нека се кључеви налазе у распону $(0, n)$ тада је нам је потребно $s = \log n$ цифара да би имали могућност да „направимо“ број n. У том случају копмлексност је $O(n \log n)$. Ако тј. $O(n \log k)$, где је k максимална вредност кључа. Прича на ову тему би могла даље да се настави и уопшти, али ћемо се за сад зауставити овде.

Heap sort

Heap sort је алгоритам за сортирање који по комплексности припада класи $O(n \log n)$ (у најгорем случају). Оно што је његова предност јесте што не захтева гломазне рекурзије, нити додатни меморијски простор. Одличан је за примену на листама са огромним количинама података. Као што му име каже овај алгоритам за свој рад користи heap (гомила) структуру.

Heap представља бинарно стабло са особином да је кључ корена већи од кључева потомака (и оба подстабла имају својство heap-а). Ако је уз то стабло комплетно (сваки лист, односно чвор који нема својих потомака, је на истом растојању од корена[‡]) могуће је његово смештање у листу, без непотребног трашења простора, односно „рупа“ у листи. Како? Ако нумеришемо чворове почев од 1 у корену стабла, левог потомка чвора k нумеришемо са $2k$, а десног са $2k + 1$, тада се heap може сместити у узастопним позицијама у листу тако ће се чвор нумерисан са k наћи на k-том месту у листи.

Heap алгоритам ради тако што од улазних података креира heap који смешта у листу. Након тога, највећи елемент, који се захваљујући конструкцији листе налази на почетку, смешта на почетак излазне листе, избацује га из „улазне“ листе чувајући, притом, својство heap-а. Поступак се понавља све док има елемената на heap-у. Овакав опис подразумева да вам је за реализацију алгоритма потребан додатни меморијски простор, јер се листа са heap-ом и излазна листа одвојено чувају. Да би се дуплирање избегло, могуће је обе листе сместити у једну константне дужине. Наиме, у првом делу „заједнике“ листе се чува heap, док се у остатку налази сортирани део. Када се један елемент дода сортираном делу, он ће већ бити избачен из heap-а, чиме се дужина листе не мења и увек је једнака броју улазаних података, тј. n.

У наставку текста дајемо детаљнији опис имплементације неких од ових алгоритама.

ИМПЛЕМЕНТАЦИЈА

***Тијана Ђукић*, II$_2$, Прва крагујевачка гимназија**
***Зоран Д. Васиљевић*, професор, Прва крагујевачка гимназија**

Counting sort

```
program CountingSort;
var a,b,c:array [1..100] of integer;
i,j,k,n:integer; begin
 read(n,k);
 for i:=1 to k do c[i]:=0;
 for j:=1 to n do begin read(a[j]);c[a[j]]:=c[a[j]]+1; end;
 for i:=2 to k do c[i]:=c[i]+c[i-1];
 for j:=n downto 1 do begin b[c[a[j]]]:=a[j]; c[a[j]]:=c[a[j]]-1; end;
 for i:=1 to n do write(b[i]:7);
end.
```

Овај алгоритам сортира низ природних бројева, при чему бројеви имају вредности до k, а има их n. Низ a је низ који се задаје на улазу, онај низ који треба сортирати. У низу b се памти сортиран низ a, а $c[i]$ је број појављивања броја i у низу a у другој петљи, а у трећој петљи $c[i]$ је последње место у низу где се појављује број i. Прва `for` петља иницијализује низ c тако да вредност свих његових чланова буде 0. Друга `for` петља формира низ c, односно броји колико пута се сваки члан низа $a[i]$ појављује и ту вредност памти на $a[i]$-том месту у низу c. Трећа `for` петља сабира у i-ти члан низа c све оне чланове пре њега, чиме се постиже да $c[i]$ сада показује на којем ће се месту последњи пут појавити i. Четврта `for` петља формира низ b, тако што се, најпре, на $c[i]$-то место у низ b постави тај број i, па пошто је оно сада попуњено, $c[i]$ се смањује за један. Поступак се понавља док се не формира цео низ b.

Heap sort

```
program HeapSort;
var a:array [1..100] of integer;
    i,n,h,s,f,l:integer;
begin
 read(n);
 for i:=1 to n do read(a[i]);
 for i:=2 to n do
  begin
   h:=a[i]; s:=i;f:=s div 2;
   while ((s>1) and (a[f]<h)) do
   begin
     a[s]:=a[f]; s:=f; f:=s div 2;
   end;
 a[s]:=h;
end;
```

```
for i:=n downto 2 do
 begin
 l:=a[i]; a[i]:=a[1]; f:=1;
 if ((i-1>=3) and (a[3]>a[2]))
    then s:=3 else s:=2;
 while (s<=i-1) and (a[s]>l) do
    begin
    a[f]:=a[s]; f:=s; s:=2*f;
    if ((s+1<=i-1) and (a[s+1]>a[s]))
        then s:=s+1;
    end;
  a[f]:=l;
 end;
for i:=1 to n do write(a[i]:9);
end.
```

Прва `for` петља формира heap стабло, друга уклања вредност корена, ставља је на почетак сортираног дела низа и онда нађе место за последњи члан несортираног дела низа тако да несортирани део задржи heap структуру.

Променљива s представља индекс потомка, а f индекс родитељског чвора. Први члан низа сам за себе јесте heap, зато прва `for` петља почиње од 2. За свако i се вредност i-тог члана (који се умеће) запамти у променљивој h ($h := a[i]$), а затим индекс потомка узима вредност $s := i$, док индекс родитеља $f := i \operatorname{div} 2$. Тражи се место за h, члан који треба да се убаци. То се обавља кроз `while` петљу. У оквиру `while` се на место потомка поставља вредност родитеља ($a[s] := a[f]$), затим индекс потомка узима вредост индекса родитеља ($s := f$). Индекс родитеља новог потомка је $f := s \operatorname{div} 2$. Вредности у heap-у се „спуштају“ низ heap на релацији потомак-родитељ а h се практично „пење“ уз heap, све док је h веће од тренутног оца и док се није стигло до корена ($(a[f] < h) \text{ and } (s > 1)$). Из `while` петље могуће је изаћи ако није $h > a[f]$ или није $s > 1$. У првом случају (није $h > a[f]$) h се смешта на место са индексом s, а у другом (није $s > 1$) у корен, први члан низа a, који такође има индекс s, дакле $a[s] := h$. Након завршетка прве `for` петље формиран је heap.

Друга `for` петља почиње од n и иде до 2. На l се поставља вредност последњег члана heap-а ($l := a[i]$), а на његово место вредост корена ($a[i] := a[1]$) чиме се сортирани део низа a повећава за 1. Низ a је сада сортиран од i-те позиције до n-те позиције (краја). Део низа a од позиције 1 до позиције $i - 1$ треба преуредити, сместити l, тако да има структуру heap-а. Пре уласка у `while` петљу, у којој се тражи место за l, индекс родитеља (f) се постави на 1. Дакле, креће се од корена, а онда се одреди који је од потомака 2 и 3 већи и за индекс потомка (s) се постави индекс већег. У `while` петљи се на место родитеља постави вредност потомка ($a[f] := a[s]$). Затим, индекс родитеља узима вредост индекса потомка ($f := s$) а индекс потомка тог новог родитеља је индекс већег од потомака ($s := 2f$ или $s := 2f + 1$). Вредности у heap-у се „пењу“ уз heap на релацији родитељ-потомак, док се l „спушта“ низ heap, све док је l мање од тренутног потомка и док се није стигло до краја heap-а ($(a[s] > l) \text{ and } (s < i - 1)$). Изласком из `while` петље одређен је индекс f где треба поставити вредност l ($a[f] := l$). Успостављен је heap. Након извршавања друге `for` петље низ је сортиран у неопадајући.

Bubble sort

```
poz:=n; repeat
     d:=poz;poz:=0;
     for i:=1 to d-1 do
           if a[i]>a[i+1] then
         begin
              p:=a[i];a[i]:=a[i+1];
              a[i+1]:=p;
              poz:=i;
         end;
until poz=0;
```

Insertion sort

```
for i:=2 to n do
     begin
       p:=a[i];j:=i-1;
       while (j>0) and (a[j]>p) do
          begin
            a[j+1]:=a[j];j:=j-1;
          end;
        a[j+1]:=p;
     end;
```

Статијата прв пат е објавена во списанието Тангента на ДМ на Србија во 2004/05 година