

In-place a in situ algoritmy

I.Kolingerová

Obsah:

1. In-place a in situ algoritmy
2. Výhody in-place a in situ
3. Příklady
4. In place sorting



1. In-place a in situ algoritmy

- Algoritmy pro vstupní data uložená v poli, sice se vejdou do paměti, ale není místo na jejich uložení do složitějších datových struktur
- **In place:** kromě vstupu v poli povolena jen $O(1)$ dodatečná paměť
- **In situ:** kromě vstupu v poli povolena jen $O(\log n)$ dodatečná paměť
- Vstup během chodu programu obvykle přepsán výstupem – není nutný ani postačující rys – výstup může být konstantní velikosti
- Výstup typicky uložen jako permutace vstupu

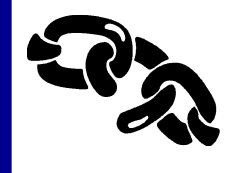
2

Př.: Otočení řetězce

```
function reverse (a[0..n])  
  allocate b[0..n]  
  for i from 0 to n  
    b[n-i] = a[i]  
return b
```

x

```
function reverse_in_place (a[0..n])  
  for i from 0 to floor(n/2)  
    swap (a[i],a[n-i])
```



3

Určitá nejednoznačnost, co počítat do velikosti paměti:

- Pro „malá“ data - pointer $O(1)$ paměť
- Pro „velká“ data - pointer potřebuje $O(\log n)$ bitů pro specifikaci indexu/adresy do velikosti n – obvykle se toto ale ignoruje

4

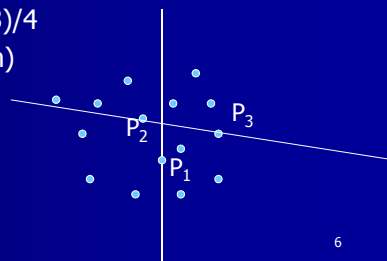
2. Výhody in-place a in situ

- Jde zpracovat větší dat. množiny
- Větší lokalita reference – vhodné pro paměť. hierarchie
- Méně náchylné k selhání – nevyžaduje velké objemy paměti, kt. nemusí v době běhu být dostupná
- Hodně velké dat. množiny často na discích – pomalý „náhodný přístup“, „levná“ dodatečná paměť, lokalita reference pak důležitější než objem dodatečné paměti

5

3. Příklady

- Jeden z hlavních triků: ukládat pointery implicitně permutací bloku dat
- Např. **Willardův 2D dělicí strom** - n static. bodů rekurzivně rozděleno do 4 oblastí 2 přímkami:
 - První - svislá bodem p_1 s mediánem v x
 - Druhá - body p_2, p_3 voleny tak, aby počet bodů v každé oblasti byl přesně $(n - 3)/4$ (Ize - tzv. ham-sandwich teorém)
 - zabírá $O(n)$



6

Willardův 2D dělicí strom

- **Prostorově efektivní verze:** p_1, p_2, p_3 v čele pole, zbytek rozdělen na 4 podpole stejné velikosti, na nich rekurze => 0 prostor navíc
- Prohledávání v sublineárním čase, $O(1)$ místo, předzprac. $O(n \log n)$
- Možná i semidynamizace (=vkládání)



7

Př.: Prohodíte 2 posloupnosti v lineárním čase, s užitím $O(1)$ dodatečné paměti

A=LSP, LSP ->PSL

(posloupnosti L,S,P mohou mít různou délku => problém)

1. Zrcadlově obrát' L,S,P
2. Zrcadlově obrát' celé A

Př.: 1 2 3 4 | 5 6 7 | 8 9 10 11 12

Ad 1) 4 3 2 1 | 7 6 5 | 12 11 10 9 8

Ad 2) 8 9 10 11 12 | 5 6 7 | 1 2 3 4

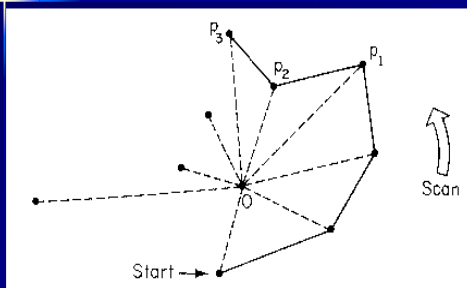
8

Př.: In-place konvexní obálka

- Aplikace prost. efekt. verze - vrcholy $CH(S)$ - prefix téhož pole
- Úprava Grahamova algoritmu: potřebuji In-place sort

9

Grahamovo prohledávání



Kritérium vyřazení bodu:
Úhel $p_1 p_2 p_3 \geq \pi$ (pravotočivost)

Start: extrémní bod
(nejpravější bod
s nejmenší souř. y)

1. $p_1 p_2 p_3$ je pravotočivý:
Eliminuj vrchol p_2 a
zkontroluj $p_0 p_1 p_3$
2. $p_1 p_2 p_3$ je levotočivý:
Zkontroluj $p_2 p_3 p_4$

10

Graham_In_Place_Scan

S - vstup. body, n - počet, d = 1 - sort rostoucí, horní $CH(S)$,
-1 - sort klesající, dolní $CH(S)$

```
1. Graham_In_Place_Sort (S,n,d)
2. h = 1
3. for i := 1 to n-1 do
4.   while (h >= 2) and not right_turn(S[h-2],S[h-1],S[i])
5.     h := h-1 { pop top element from the stack }
6.   end while
7.   swap S[i] and S[h]
8.   h := h + 1
9. end for
10. return h
```

Vrací dolní nebo horní $CH(S)$ v $S[0]$ až v $S[h-1]$,
horní $CH(S)$ uložena CW zleva doprava,
dolní CW zprava doleva

11

Graham_In_Place_Hull (S,n)

S - vstup. body, n - počet, d = 1 - sort rostoucí, horní $CH(S)$,
-1 - sort klesající, dolní $CH(S)$

```
1. h := Graham_In_Place_Scan(S,n,1) // O(n log n)
2. for i := 0 to h-2 do // O(h)
3.   swap S[i] and S[i+1]
4. end for
5. h' := Graham_In_Place_Scan (S+h-2,n-h+2,-1)
6. return h+h' - 2
```

Celkem $O(n \log n)$ čas, $O(1)$ extra paměť

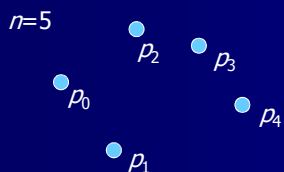
Vylepšení: nalézt extrémny, rozdělit pole na 2 části -
- pro horní a dolní $CH(S)$, pak každý bod
jen v 1 volání Scan

Větší vylepšení: pomocí MergeSort

12

Konvexní obálka - příklad (1)

Po setřídění podle x:



S:

p_0	p_1	p_2	p_3	p_4	...
p_0	p_2	p_1	p_3	p_4	...
p_0	p_2	p_3	p_1	p_4	...
p_0	p_2	p_3	p_4	p_1	...

h	i	Akce
1	1	Swap S_1, S_1 ☺
2	2	kontrola S_0, S_1, S_2 (p_0, p_1, p_2) , dec(h)
1	2	Swap S_2, S_1
2	3	kontrola S_0, S_1, S_3 (p_0, p_2, p_3) , OK, swap S_2, S_3
3	4	kontrola S_1, S_2, S_4 (p_2, p_3, p_4) , OK, swap S_3, S_4

3

Konvexní obálka - příklad (2)

Po výpočtu horní CHS:

levý bod	horní konvexní obálka	pravý bod	...
----------	-----------------------	-----------	-----

Pak přesun levého bodu doprava:

horní konvexní obálka	pravý bod	levý bod	...
-----------------------	-----------	----------	-----

Po výpočtu dolní konvexní obálky:

horní konvexní obálka	pravý bod	dolní konvexní obálka (uložena obráceně)	levý bod	...
-----------------------	-----------	---	----------	-----

Výstup:

Konvexní obálka	
-----------------	--

14

5. In-place a in situ sorting

a) **spojité pole** – konst. čas na přístup a prohození, dlouhý čas na posuvy

Pokud ignorujeme $O(\log n)$ na pointery:

- Heapsort – ano, konst. dodatečná paměť
- Quicksort – in situ, $O(\log n)$ paměť na rekurzi (\leq hloubky stromu); v nejhorším případě potřebuje $O(n^2)$ čas, pak $O(n)$ dodat. paměť na rekurzi, ale velmi nepravděpodobně
- Mergesort – ne

15

b) **zřetěz. seznamy** – vyhledání podle indexu $O(n)$

- Quicksort a heapsort – nutné drastické modif., aby alesp. $O(n^2)$; zřetězené seznamy pro tento druh algoritmů nejsou moc vhodné
- Mergesort – $O(n \log n)$ čas, $O(\log n)$ dodat. paměť, krok Merge se zřetězenými seznamy lehčí než s poli; Mergesort s poli – dobrá lokality reference, pro data na disku lepší než jiné alg.

16

MergeSort (A,l,h)

```
// Seřadí pole A[l..h] vzestupně, předp.  $h > l \geq 0$ 
```

```
if  $h > l$   
   $m \leftarrow \text{int}((l+h)/2)$   
  MergeSort (A,l,m);  
  MergeSort (A,m+1,h);  
  Merge(A,l,m,h)  
endif
```

Merge – obyčejná verze využívá pole navíc pro snadné spojení dílčích polí, in place verze musí udělat v poli A

17

Merge (A,l,m,h)

```
// Spojí 2 pole A[l..m] a A[m+1..h] seřazená vzestupně,  
obyčejná verze – užije dočasný buffer Save[l..h]
```

```
next  $\leftarrow l$ ;  $i \leftarrow l$ ;  $j \leftarrow m+1$ ;  
while ( $i \leq m$ ) and ( $j \leq h$ ) do  
  if  $A[i] > A[j]$  then Save[next]  $\leftarrow A[j]$ ; inc (j)  
  else Save[next]  $\leftarrow A[i]$ ; inc(i)  
  endif;  
  inc (next)  
endwhile  
if  $i \leq m$  then Save[next..h]  $\leftarrow A[i..m]$   
  else Save[next..h]  $\leftarrow A[j..h]$   
A[l..h]  $\leftarrow$  Save[l..h]
```

$O(n \log n)$, ale paměť navíc, kopírování zdržuje

18

Merge (A,l,m,h)

```
// Spojí 2 pole A[l..m] a A[m+1..h] seřazená vzestupně, in
// place verze – žádný dočasný buffer
i <- l; j <- m+1;
while (i <=m) and (j<=h) do
  if A[i] <= A[j] then inc (i) // výběr z levého pole
  else
    // výběr zprava – rotace mezi i, j-1 o 1 místo doprava
    tmp <- A[j];
    Posun_pole_vpravo (A,i,j-1) ;
    A[i] <- tmp;
    inc (i); inc (m); inc (j); // vše se posunulo doprava
  endif;
endwhile
// Vše, co zbývá v A[j..h] , je na místě
```

19

Literatura

- http://en.wikipedia.org/wiki/In-place_algorithm
- W.-K.Hon: Tutorial,
<http://www.cs.nthu.edu.tw/~wkhon/algo08-tutorials/tutorial1b.pdf>
- H.Bronnimann, J.Iacono et.al.: Space-efficient planar convex hull algorithms,
<http://photon.poly.edu/~hbr/publi/inplace-ch2d/inplace-tcs03.pdf>
- H.Bronnimann, T.M.Chan, E.Y.Chen: Towards In-Place Geometric Algorithms -and Data Structures,
<http://photon.poly.edu/~hbr/publi/inplace-ch3d/talg.06.pdf>
- Většinou popis konkrétních algoritmů

20