Algoritmi hibrizi de sortare

Butcan Vlad

Universitatea Politehnica Bucuresti

FacultateadeAutomaticasiCalculatoare

1 Introducere

1.1 Descriere Algoritmi Hibdrizi de Sortare

Algoritmii hibrizi de sortare folosesc doi sau mai mulți algoritmi simpli sau hibrizi de sortare. Pentru problema aleasă folosesc citirea din fișier de pe prima linie a unui număr N. N reprezintă mărimea vectorului ce urmează a fi sortat și numărul de valori aflate pe următoarele linii din fișier.

1.2 Aplicatii practice

Algoritmii hibrizi de sortare sunt folosiți în mare plan în motoarele de căutare web, în limbajul de programare Python și în OpenJDK. De asemenea sunt folosiți și de Apple Swift Standard Library 5. În principiu algoritmii hibrizi de sortarea se folosesc deoarece în lumea reală există un număr mare de date în care foarte des se întâlnesc bucăți de date aranjate corect în ordinea sa.

1.3 Algoritmi aleși

TimSort

IntroSort

1.4 Criterii de evaluare

Criteriul dupa care se vor evalua algoritmii IntroSort și TimSort va fi timpul de runtime a programului doar în momentul în care acesta apelează funcția propiu-zisă de sortare. Toate rezultatele de runtime se vor salva în folderul ./out în fișierul runtimeResult.txt. De asemenea in acest fisier sunt reprezentatea datele precum best runtime, edium runtime și worst runtime. Pentru a evalua diferența pentru acești 2 algoritmi voi prezenta în continuare câteva grafice și tabele.

2 Prezentarea soluțiilor

2.1 Descrierea modului de funcționare a algoritmilor

TimSort Timsort a fost conceput pentru a profita de execuțiile de elemente ordonate consecutive care există deja în majoritatea datelor din lumea reală, execuții naturale. Se repetă peste elementele de colectare a datelor în execuții și simultan pune acele execuții într-o stivă. Ori de câte ori rulările din partea de sus a stivei se potrivesc cu un criteriu de îmbinare , acestea sunt îmbinate. Aceasta continuă până când toate datele sunt parcurse; apoi, toate rulările sunt îmbinate câte două și rămâne doar o singură rulare sortată. Avantajul îmbinării curselor ordonate în loc de îmbinare sub-liste cu dimensiuni fixe (așa cum se face prin mergesort tradițional) este că scade numărul total de comparații necesare pentru sortarea întregii liste.

Fiecare rulare are o dimensiune minimă, care se bazează pe dimensiunea intrării și este definită la începutul algoritmului. Dacă o rulare este mai mică decât această dimensiune minimă de rulare, sortarea prin inserare este utilizată pentru a adăuga mai multe elemente la rulare până când este atinsă dimensiunea minimă a rulării.

Timsort se străduiește să efectueze îmbinări echilibrate (o îmbinare îmbină astfel execuții de dimensiuni similare).

Pentru a obține stabilitatea sortării, sunt îmbinate numai execuțiile consecutive. Între două execuții neconsecutive, poate exista un element cu aceeași cheie în interiorul execuțiilor. Fuzionarea acestor două rulări ar schimba ordinea cheilor egale.

IntroSort IntroSort este algoritmul folosit de swift pentru a sorta o colecție. Introsort este un algoritm hibrid inventat de David Musser în 1993 cu scopul de a oferi un algoritm de sortare generic pentru biblioteca standard C++. Implementarea clasică a introsort se așteaptă la un Quicksort recursiv cu fallback la Heapsort în cazul în care nivelul de adâncime a recursiunii a atins un anumit max. Maximul depinde de numărul de elemente din colecție și de obicei este de 2 * log(n). Motivul din spatele acestui fallback" este că, dacă Quicksort nu a reușit să obțină soluția după 2 * log(n) recursiuni pentru o ramură, probabil că a lovit cel mai rău caz și se degradează la complexitatea O(n^2). Pentru a optimiza și mai mult acest algoritm, implementarea swift introduce un pas suplimentar în fiecare recursivitate în care partiția este sortată folosinf InsertionSort dacă numărul partiției este mai mic de 20.

2.2 Analiza complexității soluțiilor

TimSort În cel mai rău caz , Timsort ia O(n * log(n)) comparații pentru a sorta o matrice de n elemente. În cel mai bun caz, care apare atunci când intrarea este deja sortată, rulează în timp liniar, ceea ce înseamnă că este un algoritm de sortare adaptiv. Este superior lui Quicksort pentru sortarea referințelor de obiecte sau a indicatorilor, deoarece acestea necesită o indirectă costisitoare a

memoriei pentru a accesa date și a efectua comparații, iar beneficiile Quicksort privind coerența cache-ului sunt mult reduse.

IntroSort În quicksort, una dintre operațiunile critice este alegerea pivotului: elementul în jurul căruia este împărțită lista. Cel mai simplu algoritm de selecție pivot este să ia primul sau ultimul element al listei ca pivot, provocând un comportament slab pentru cazul intrării sortate sau aproape sortate. Varianta lui Niklaus Wirth folosește elementul de mijloc pentru a preveni aceste apariții, degenerând la $O(n^2)$ pentru secvențele inventate. Algoritmul de selecție pivot median-of-3 preia mediana primului, mijlociu și ultimului elemente ale listei; cu toate acestea, chiar dacă acest lucru funcționează bine pe multe intrări din lumea reală, este totuși posibil să se creeze o listă de ucigași mediană de 3 care va provoca încetinirea dramatică a unui sortare rapidă bazată pe această tehnică de selecție pivot.

2.3 Prezentarea principalelor avantaje și dezavantaje

TimSort Timsort a fost conceput pentru a profita de execuțiile de elemente ordonate consecutive care există deja în majoritatea datelor din lumea reală, execuții naturale. Avantajul principal (așa cum se face prin mergesort tradițional) este că scade numărul total de comparații necesare pentru sortarea întregii liste. Dezavantajul provine din modul galop în timpul îmbinării. În unele cazuri, modul de galop necesită mai multe comparații decât o simplă căutare liniară . Conform benchmark-urilor realizate de dezvoltator, galoparea este benefică doar atunci când elementul inițial al unei alergări nu este unul dintre primele șapte elemente ale celeilalte curse. Aceasta implică un prag inițial de 7.

IntroSort Algoritmul IntroSort se bazează pe Quicksort, Heapsort și Insertion sort motiv din care principalul avantaj este reprezentat de InsertionSort. Acesta prezintă o performanță bună atunci când avem de-a face cu o listă mică. Sortarea prin inserare este un algoritm de sortare în loc, astfel încât necesarul de spațiu este minim. Dezavantajul este că nu funcționează la fel de bine ca ceilalți algoritmi de sortare atunci când dimensiunea datelor devine mai mare.

3 Evaluare

3.1 Descrierea modalitații de construire a setului de teste

Fișiere de test sunt create în urma rulării comenzii make prin intermediul unui script care creează la alegerea utilizatorului numărul de teste și numărul de valori din acestea. Valorile din teste sunt create prin intermediul funcției random. Pentru a crea toate testele diferite, funcția random folosește la fiecare test un seed diferit. Acest seed nu se repetă niciodată precum pentru fiecare

test seedul său este egal cu numarul testului pe care îl creează. Valorile din fișierul de test pot fi între 1 și 100000 conform programului GenerareTeste.c, însă acesta poate fi modificat la preferința utilizatorului. De asemenea pentru a efectua o evaluare rapidă de verificare a corectitudinii sortării putem folosi comanda make result care ruleaza fișierul verifyProgramm.c. Programul dat verifică corectitudinea sortării fiecărui testX.out și prezintă aceste rezultate în fișierul ./out/verifyResult.txt în formatul: Testx passed/failed.

3.2 Specificațiile sistemului de calcul

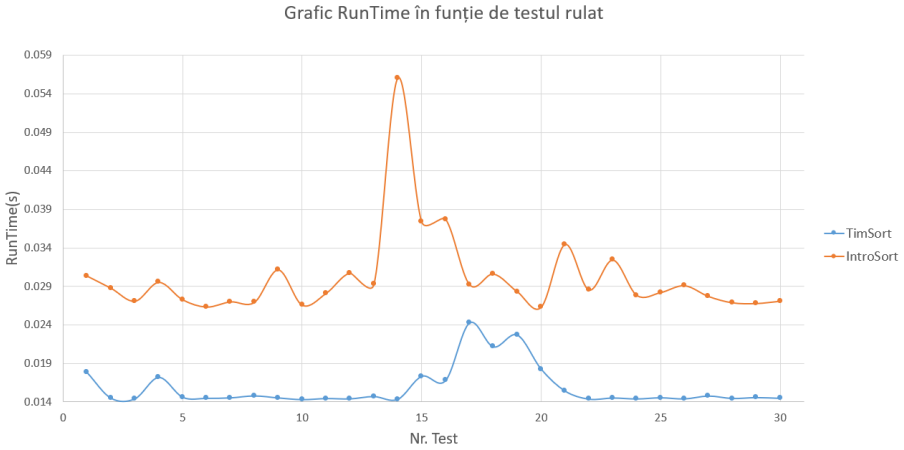
- Procesor: Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz 1.80 GHz
- Memorie RAM: 8.00 GB
- Placă Video: NVIDIA GeForce MX150 2GB RAM

3.3 Ilustrarea, folosind grafice/tabele, a rezultatelor evaluării soluțiilor

Tabelul runtime a 30 de teste a cate 100.000 valori fiecare

	A	B	C	D	E
1	TimSort			IntroSort	
2	Tests	RunTime		Tests	RunTime
3	1	0.017824		1	0.030359
4	2	0.014495		2	0.028783999
5	3	0.014421		3	0.027105
6	4	0.017201001		4	0.029549001
7	5	0.014612		5	0.027286001
8	6	0.014479		6	0.026354
9	7	0.014543		7	0.027024999
10	8	0.014793		8	0.026953001
11	9	0.014534		9	0.031163
12	10	0.014311		10	0.026594
13	11	0.014454		11	0.028122
14	12	0.014409		12	0.030750001
15	13	0.014713		13	0.029273
16	14	0.01435		14	0.056007002
17	15	0.017344		15	0.037379999
18	16	0.016793		16	0.037730999
19	17	0.024286		17	0.029259
20	18	0.021168999		18	0.030627999
21	19	0.022719		19	0.028329
22	20	0.018276		20	0.026322
23	21	0.015426		21	0.034464002
24	22	0.014372		22	0.028557001
25	23	0.014523		23	0.032441001
26	24	0.014399		24	0.027868999
27	25	0.014553		25	0.028228
28	26	0.014407		26	0.029114
29	27	0.014779		27	0.027727
30	28	0.014462		28	0.026905
31	29	0.014577		29	0.026807999
32	30	0.01448		30	0.027109001

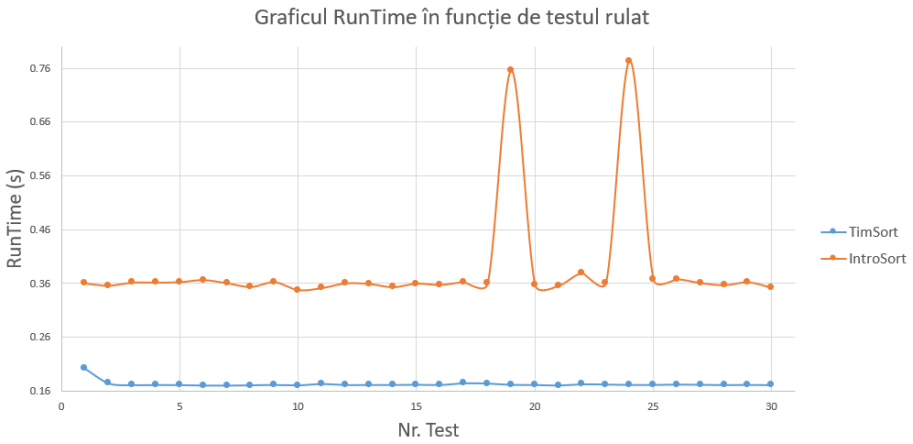
Graicul runtime a 30 de teste a cate 100.000 valori fiecare



Tabelul runtime a 30 de teste a cate 1.000.000 valori fiecare

	A	B	C	D	E
1		TimSort			IntroSort
2	1	0.202181995		1	0.360219002
3	2	0.174399003		2	0.35600701
4	3	0.171095997		3	0.361743987
5	4	0.171409994		4	0.36174199
6	5	0.171251997		5	0.362565011
7	6	0.170358002		6	0.366468012
8	7	0.170303002		7	0.360545993
9	8	0.170843005		8	0.353673011
10	9	0.171553999		9	0.36243999
11	10	0.1708		10	0.347880989
12	11	0.173262998		11	0.351583004
13	12	0.171330005		12	0.360372007
14	13	0.171452999		13	0.359299004
15	14	0.171395004		14	0.353287995
16	15	0.171918005		15	0.359914005
17	16	0.171377003		16	0.357437998
18	17	0.174591005		17	0.362470001
19	18	0.174143001		18	0.359993994
20	19	0.17162099		19	0.755177021
21	20	0.171526998		20	0.357834011
22	21	0.170361996		21	0.356128007
23	22	0.172784001		22	0.379231006
24	23	0.172244996		23	0.361530989
25	24	0.171498999		24	0.773685992
26	25	0.171445996		25	0.367698014
27	26	0.172261		26	0.368218005
28	27	0.171770006		27	0.360915989
29	28	0.171206996		28	0.35690099
30	29	0.171611995		29	0.362385988
31	30	0.171140999		30	0.352245986

Graicul runtime a 30 de teste a cate 1.000.000 valori fiecare



Analiza proprie Pentru o statistică mai reușită am decis să rulez scriptul pentru 1000 de fișiere a câte 1.000.000 de valori fiecare.
Rezultatul pentru algoritmul TimSort este următorul:

```
/mnt/c/Users/Vlad/Desktop/UPB/AA/Tema1
❯ cat out/runtimeResult.txt | tail -3
Best runTime: 0.170185000
Medium runTime: 0.181577250
Worst runTime: 1.580057979
```

Rezultatul pentru algoritmul IntroSort este următorul:

```
/mnt/c/Users/Vlad/Desktop/UPB/AA/Tema1
❯ cat out/runtimeResult.txt | tail -3
Best runTime: 0.347508013
Medium runTime: 0.381528020
Worst runTime: 3.109447002
```

Pentru a veirifica cum reacționează algoritmii pe mărimi mai mici am făcut următoarele analize:
Rezultatul pentru 1000 de teste a câte 100 valori

```
How much test do you want?
1000
Give size for all arrays
100

 /mnt/c/Users/Vlad/Desktop/UPB/AA/Tema1
❯ make run-p1
./timSort > ./out/runtimeResult.txt

 /mnt/c/Users/Vlad/Desktop/UPB/AA/Tema1
❯ cat out/runtimeResult.txt | tail -3
Best runTime: 0.000006000
Medium runTime: 0.000010842
Worst runTime: 0.000073000

 /mnt/c/Users/Vlad/Desktop/UPB/AA/Tema1
❯ make run-p2
./introSort > ./out/runtimeResult.txt

 /mnt/c/Users/Vlad/Desktop/UPB/AA/Tema1
❯ cat out/runtimeResult.txt | tail -3
Best runTime: 0.000013000
Medium runTime: 0.000019645
Worst runTime: 0.000265000
```

Rezultatul pentru 1000 de teste a câte 10 valori

```
How much test do you want?
1000
Give size for all arrays
10

 /mnt/c/Users/Vlad/Desktop/UPB/AA/Tema1
❯ make run-p1
./timSort > ./out/runtimeResult.txt

 /mnt/c/Users/Vlad/Desktop/UPB/AA/Tema1
❯ cat out/runtimeResult.txt | tail -3
Best runTime: 0.000001000
Medium runTime: 0.000002419
Worst runTime: 0.000034000

 /mnt/c/Users/Vlad/Desktop/UPB/AA/Tema1
❯ make run-p2
./introSort > ./out/runtimeResult.txt

 /mnt/c/Users/Vlad/Desktop/UPB/AA/Tema1
❯ cat out/runtimeResult.txt | tail -3
Best runTime: 0.000001000
Medium runTime: 0.000002138
Worst runTime: 0.000019000
```

4 Concluzii

După cum s-a observat, în urma analizelor pe date de mărimi diferite de la 10 la 1.000.000, algoritmul TimSort este mult mai efectiv din punt de vedere al timpului de rulare. Pe un set de teste mai mare această diferență se simte foarte clar: diferența depășește 50% efectivitate runtime. Luând în considerație faptul că în ziua de azi există o cantitate foate mare de informație ce trebuie prelucrată, alegerea mea ar fi TimSort fără nici o îndoială. Consider că acesta este principalul motiv din care Python și JavaJDK folosesc la bază algoritmul TimSort.

References

1. https://en.wikipedia.org/wiki/Timsort
2. https://aquarchitect.github.io/swift-algorithm-club/Introsort/
3. https://www.geeksforgeeks.org/introsort-or-introspective-sort/
4. https://en.wikipedia.org/wiki/Introsort
5. https://www.geeksforgeeks.org/timsort/