Algoritam Timsort

Beg, Mislav

Master's thesis / Diplomski rad

2019

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Science / Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:217:099092

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-03-28**



Repository / Repozitorij:

Repository of the Faculty of Science - University of Zagreb







SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Mislav Beg

ALGORITAM *TIMSORT*

Diplomski rad

Voditelj rada:
doc. dr. sc. Vedran Čačić

Zagreb, ožujak, 2019.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Sadržaj **iii**

Uvod **2**

1 Sortiranje **3**
1.1 Opis problema 4
1.2 Algoritmi za sortiranje 9

2 Timsort **16**
2.1 Opis rada 16
2.2 Kod 27
2.3 Složenost 51

Bibliografija **57**

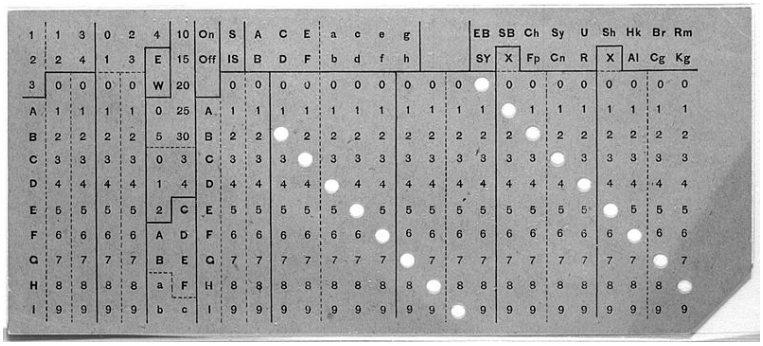
Uvod

Među prvim algoritmima s kojima se upoznajemo na nastavi računarstva su algoritmi za sortiranje. Oni nam služe kao uvod u proučavanje algoritama, te pomoću njih naučimo razne pojmove koji nam kasnije služe za analizu algoritama. Također, zato što algoritama za sortiranje ima puno, na njima možemo analizirati i pokazati različite tipove algoritama — poznati primjer toga je algoritam *MergeSort*, koji je tipa „podijeli pa vladaj". No algoritmi za sortiranje nisu privukli pažnju samo na nastavi računarstva. Dapače, od početka računarske znanosti algoritmi za sortiranje su privukli veliku pozornost znanstvenika. Prvi strojevi za sortiranje razvijeni su već u 19. stoljeću. Herman Hollerith je izumio prvi takav stroj — elektromehanički tabulator (slika 1) — da bi američkoj vladi ubrzao proces popisa stanovništva (skupljanja i obrade informacija o stanovništvu) za koji je 1880. godine, prije korištenja Hollerithove naprave za sortiranje, trebalo osam godina. Naprava je koristila probušene kartice (slika 2) za obradu podataka. Ideja je bila da se određeni podaci mogu prikazati tako da provjerimo postoji li na nekom mjestu na kartici rupa ili ne. Na primjer, ako na kartici imamo mjesto koje označava je li osoba u braku, rupa u kartici na tom mjestu može označavati da osoba je u braku, dok bi izostanak rupe označavao da osoba nije u braku. Elektromehanički tabulator Hermana Holleritha je postao i temelj za *RadixSort* — prvi poznati algoritam za sortiranje. S početkom razvoja modernih računala intenzivno su se počeli razvijati i stvarati algoritmi za sortiranje, te se zbog njihove važnosti i korisnosti do današnjeg dana još uvijek smišljaju novi algoritmi.

U ovom radu ćemo objasniti kako radi algoritam za sortiranje *TimSort*. *TimSort* je standardni algoritam za sortiranje u Pythonu još od verzije 2.3. Uz to ćemo na kanonskoj implementaciji *TimSorta* pokazati kako radi u praksi, te ćemo staviti *Timsort* u kontekst ostalih algoritma za sortiranje koje ćemo ukratko opisati.



Slika 1: Elektromehanički tabulator Hermana Holleritha. Izvor: [1]



Slika 2: Probušena kartica za sortiranje. Izvor: [1]

Poglavlje 1

Sortiranje

Postoji mnogo razloga zašto nam se isplati sortirati podatke. Nabrojit ćemo neke od najčešćih upotreba sortiranih podataka.

- Pitanje jedinstvenosti elemenata — postoje li duplikati u danom skupu od n elemenata? Efikasni algoritam za rješavanje tog problema je da se elementi prvo sortiraju pa linearnim pretraživanjem provjerimo postoji li neki par susjednih jednakih elemenata.

- Efikasno traženje određenog podatka unutar velikog skupa podataka — državne institucije, privatne kompanije i mnoge internetske aplikacije moraju baratati s ogromnim količinama podataka te je često tim podacima potrebno više puta pristupiti. Ako su ti podaci sortirani, traženim podacima možemo pristupiti jednostavno i brzo. Traženi elementi unutar sortiranog niza duljine n mogu se naći koristeći binarno pretraživanje u $O(\log n)$ koraka — zato su sortirani nizovi podataka prikladni kad trebamo mnogo puta brzo naći određeni podatak u nizu.

- Organizirana obrada podataka — podaci koji su sortirani nam omogućuju da ih obradimo u nekom definiranom poretku, čime je proces organiziraniji i robusniji.

Sortiranje u formi razvrstavanja je bitno i za razne industrijske procese. Na primjer, optičko sortiranje je automatizirani proces sortiranja krutih proizvoda pomoću kamera i lasera te ima široku uporabu u industriji hrane. Najčešće se koristi u proizvodnji hrane poput krumpira, voća, povrća i orašastih plodova. Takva tehnologija se također koristi u farmaceutskoj i prehrambenoj proizvodnji, obradi duhana, recikliranju otpada i ostalim industrijama. U usporedbi s ručnim sortiranjem, koje često može biti nekonzistentno i sporo, optičko sortiranje pomaže u poboljšanju kvalitete proizvoda, maksimiziranju dobiti i povećanju prinosa, te smanjuje potrebu za ručnim radom [2].

1.1 Opis problema

Sortiranje je problem u kojem za dano polje podataka želimo naći rastući niz u kojem se nalaze točno svi elementi tog polja. Zadano nam je polje usporedivih elemenata A — s $A[i]$, odnosno a_i, označavamo i-ti element polja. Svaki element polja $A[i]$ ima svoj ključ $K[i]$ s obzirom na koji se sortira. Problem sortiranja glasi: Za dano polje A, poredaj elemente od A tako da vrijedi: ako je $K[i] < K[j]$, onda je $i < j$. Ako postoje elementi u polju A koji su jednaki, oni moraju biti susjedni, tj. ako vrijedi $K[i] = K[j]$ u sortiranom polju, tada ne postoji k tako da $i < k < j$ i $K[i] \neq K[k]$ (definicija prema [7]). Znači, sortirano polje će biti poredak elemenata polja A takav da ti elementi čine rastući niz. Prema [8], relacija uspoređivanja „<" je definirana na ključevima tako da su sljedeći uvjeti zadovoljeni za bilo koje ključeve a, b i c:

- točno jedna od mogućnosti $a < b$, $b < a$ i $a = b$ je zadovoljena;
- ako je $a < b$ i $b < c$, onda je $a < c$.

Iz ovih uvjeta slijedi da je relacija uspoređivanja „<" definirana na ključevima linearni uređaj. Iz prvog svojstva slijedi usporedivost i irefleksivnost, a iz drugog tranzitivnost uređaja.

Sortiranje n-torki se može provesti na temelju jedne ili više komponenti elemenata. Na primjer, ako su naši podaci rezultati studenata na kolokviju, svaki podatak može sadržavati više komponenti: ocjenu, JMBAG, ime studenta, ... Od tih komponenti jednu možemo odabrati kao ključ za sortiranje, npr. ocjenu. U sortiranom polju, studenti s nižom ocjenom će se nalaziti ispred studenata s višom ocjenom. Novi ključ za sortiranje se može napraviti od dva ili više ključeva, po leksikografskom uređaju. Prvi ključ se onda zove primarni ključ za sortiranje, drugi se zove sekundarni ključ, ... Na primjer, s obzirom da će neki studenti imati iste ocjene, kao sekundarni ključ bismo mogli odabrati ime studenta — tada će se poredak studenata s istom ocjenom odrediti leksikografskim uređajem na temelju njihovih imena.

Da bismo lakše raspravljali o svojstvima algoritama za sortiranje, definirat ćemo neke pojmove kojima opisujemo te algoritme, te pojmove vezane uz analizu algoritama:

- *Stabilni algoritmi*: održavaju originalan relativni poredak podataka s istim ključem, tj. ako u originalnom polju imamo podatke s istim ključem, njihov relativni poredak (tj. koji od njih nastupa prije drugog) mora biti isti i u sortiranom polju.
- *Prilagođavajući algoritmi*: algoritmi kojima na vrijeme izvršavanja utječe činjenica da je dio ulaznih podataka već bio sortiran prije početka rada algoritma.
- *Inplace algoritmi*: koriste količinu dodatne memorije za sortiranje koja ne ovisi o broju podataka koji se sortiraju.

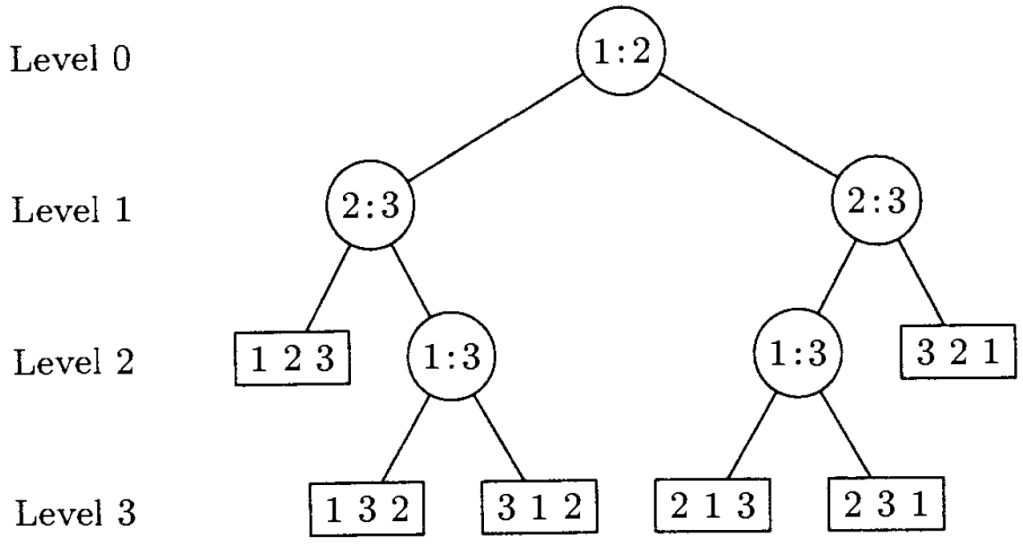
- *Uspoređujući algoritmi*: algoritmi za sortiranje koji sortiraju podatke samo pomoću relacije usporedbe koja određuje koji od dva podatka bi trebao prvi nastupiti u finalnom sortiranom nizu. Budući da *Timsort* pripada ovoj vrsti algoritama, u ovom radu ćemo se usredotočiti uglavnom na takve algoritme. Iz teorije znamo da je za ovakvu vrstu algoritama prosječna vremenska složenost odozdo ograničena — što je upravo tvrdnja teorema 1.1.1. Algoritmi koji ne koriste samo relaciju uspoređivanja ključeva elemenata mogu imati i bolje performanse — ali samo uz određene uvjete. Primjer takvog algoritma dat ćemo kasnije.

- *Vremenska složenost*: funkcija koja nam govori koliki je otprilike broj usporedbi koje algoritam mora napraviti da bi sortirao podatke. Obično promatramo vremensku složenost u najgorem, prosječnom i najboljem slučaju. Tipično je najteže procijeniti prosječnu složenost jer se ona temelji na statističkim tehnikama i procjenama. Za algoritam sortiranja je tipično da je „dobra" prosječna složenost $O(n \log n)$, a „loša" $O(n^2)$, pri čemu je n broj podataka koje treba sortirati.

  Sada ćemo dokazati teorem o donjoj granici za složenost uspoređujućih algoritama. Dokaz je prilagođen iz [8], odakle su preuzete i slike.

Teorem 1.1.1 (Vremenska složenost uspoređujućih algoritama)**.** *Uspoređujući algoritmi za sortiranje n podataka ne mogu imati vremensku složenost bolju od* $O(n \log n)$.

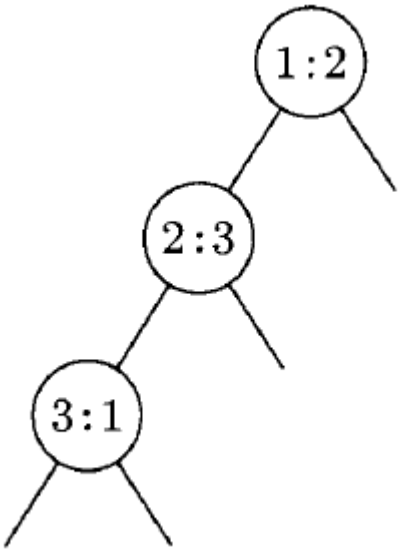
Dokaz. S obzirom da se tvrdnja teorema odnosi na uspoređujuće algoritme, jedini način na koji možemo sortirati elemente je koristeći relaciju „<". Radi jednostavnosti pretpostavljamo da su svi ključevi različiti (za jednake ključeve vidi [8, poglavlje 5.3.1., vježbe 3–12]). Za algoritme koji zadovoljavaju ove uvjete definirat ćemo *prošireno binarno stablo*. To će biti reprezentacija načina na koji ovakvi algoritmi sortiraju. Na slici 1.1 vidimo kako izgleda prošireno binarno stablo za algoritam koji sortira polje od tri elementa. Svaki unutarnji čvor (nacrtan kao krug) sadrži dva indeksa $i : j$ koji označavaju usporedbu ključeva $K[i]$ i $K[j]$. Lijevo podstablo čvora predstavlja usporedbe koje slijede ako je $K[i] < K[j]$, a desno podstablo usporedbe koje slijede u slučaju $K[i] > K[j]$. Listovi stabla (nacrtani kao pravokutnici) sadrže jednu permutaciju $\alpha_1, \alpha_2, \ldots, \alpha_n$, tj. označavaju da za ključeve vrijedi poredak $K_{a_1} < K_{a_2} < \ldots < K_{a_n}$.



Slika 1.1: Prošireno Binarno Stablo za sortiranje tri elementa

U korijenu stabla uspoređujemo $K[1]$ i $K[2]$. Ako je $K[1] < K[2]$, sljedeći korak je da uspoređujemo $K[2]$ i $K[3]$ (lijevo podstablo). Ako je $K[2] < K[3]$ opet se pomičemo u lijevo podstablo i došli smo do jednog poretka — $K[1] < K[2] < K[3]$ (pravokutni list „1 2 3" u stablu). Ako je na razini 1 $K[2] > K[3]$ treba napraviti još jednu usporedbu da ustanovimo odnos između $K[1]$ i $K[3]$. Nakon te usporedbe smo došli do još jedne permutacije ključeva. Analogni postupak se odvija u desnom podstablu korijena.

Moguće je da se naprave i suvišne usporedbe. Pogledajmo sliku 1.2. Nema potrebe uspoređivati 3:1 jer ako je $K[1] < K[2]$ i $K[2] < K[3]$, po tranzitivnosti vrijedi $K[1] < K[3]$ — u lijevom podstablu čvora 3:1 ne može se nalaziti nijedna permutacija. S obzirom da nas zanima najmanji broj mogućih usporedbi (jer pokušavamo naći donju granicu broja usporedbi), pretpostavljamo da ne postoje suvišne usporedbe. Znači, imamo prošireno binarno stablo u kojem svaki list odgovara jednoj permutaciji. Sve permutacije ulaznog polja su moguće i svaka permutacija definira jedinstven put od korijena do lista — zaključujemo da kada sortiramo polje sa n elemenata imamo točno $n!$ listova.



Slika 1.2: Primjer suvišne usporedbe

Sada prelazimo na sam dokaz. Za polje veličine n zanima nas prosječan broj usporedbi, tj. kolika je prosječna duljina puta od korijena do lista. Promotrimo opet sliku 1.1. Prosječan broj usporedbi u tom stablu je

$$\frac{2+3+3+3+3+2}{6} = 2\frac{2}{3}\,. \tag{1.1}$$

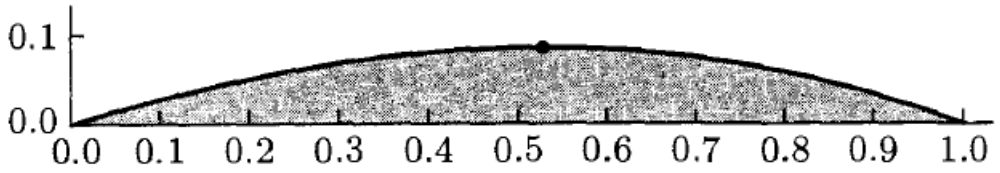
Generalno, prosječan broj usporedbi algoritma je suma udaljenosti listova od korijena (tj. *duljina puta*) podijeljena sa brojem listova — $n!$. Iz [9, poglavlje 2.3.1], znamo da se *minimalna* duljina puta u binarnom stablu s N listova postiže ako postoji $2^q - N$ listova na razini $q-1$ i $2N - 2^q$ na razini q, pri čemu je $q = \lceil \log_2 N \rceil$ (korijen je na razini 0). U tom slučaju, minimalna duljina puta je

$$(q-1)(2^q - N) + q(2N - 2^q) = (q+1)N - 2^q\,. \tag{1.2}$$

Ako q zapišemo kao $q = \log_2 N + \omega$, $0 \le \omega < 1$, formula za minimalnu duljinu puta postane

$$N(\log_2 N + 1 + \omega - 2^\omega)\,. \tag{1.3}$$

Graf funkcije $1 + \omega - 2^\omega$ je prikazan na slici 1.3.



Slika 1.3: Funkcija $1 + \omega - 2^{\omega}$

Za $0 < \omega < 1$ vrijednost funkcije je pozitivna ali mala, uvijek manja od

$$1 - (1 + \ln\ln 2)/\ln 2 \approx 0.08607134\,. \tag{1.4}$$

Dijeljenjem (1.3) sa N slijedi da je najmanji mogući prosječan broj usporedbi između $\log_2 N$ i $\log_2 N + 0.0861$. Ako stavimo da je $N = n!$, dobit ćemo donju granicu za prosječan broj usporedbi bilo kojeg uspoređujućeg algoritma. S obzirom da se logaritmi s različitim bazama razlikuju samo u multiplikativnoj konstanti nema potrebe za pretvaranjem logaritama u istu bazu. Iz sljedećih nejednakosti,

$$\log_2 n! = \log_2 1 + \log_2 2 + \ldots + \log_2 n \le n \log_2 n \tag{1.5}$$

$$\begin{aligned}\log_2 n! &\ge \log_2 \left\lfloor \frac{n}{2} \right\rfloor + \log_2 (\left\lfloor \frac{n}{2} \right\rfloor + 1) + \ldots + \log_2 n \ge \frac{n}{2} \log_2 \frac{n}{2} = \frac{1}{2} n(\log_2 n - 1) \\ &\ge \frac{1}{3} n \log_2 n, \text{ za } n \ge 2^3 = 8\end{aligned} \tag{1.6}$$

asimptotski gledano, dobijemo

$$\log_2 n! + O(1) = n \log_2 n - \frac{n}{\ln 2} + O(\log n) = O(n \log n). \tag{1.7}$$

Time je dokazana tvrdnja teorema. □

Da bismo stavili *Timsort* u kontekst, dat ćemo kratki pregled najvažnijih algoritama za sortiranje koji su se pojavljivali kroz povijest, te ćemo neke od njih opisati. Nadalje, *Timsort* je hibridni algoritam: kombinacija *MergeSorta* i *InsertionSorta*, stoga ćemo predstaviti i te algoritme.

1.2 Algoritmi za sortiranje

U tablici 1.1 možemo vidjeti razvoj algoritama za sortiranje kroz povijest. Vidimo da su mnogi poznati algoritmi nastali u početku razvoja algoritama za sortiranje: *MergeSort* i *InsertionSort* koji su i danas relevantni, te ih mnogo algoritama koristi na neki način, uključujući i *TimSort*. Dapače, ispada da su svi najpoznatiji algoritmi, mnogi od kojih se uče na Matematičkom odsjeku PMF-a, razvijeni u razmaku od dvadeset godina, od *MergeSorta* do *Heapsorta*. Vidimo da su mnogi algoritmi koji slijede, te mnogi moderni algoritmi, modificirane verzije starijih algoritama, odnosno hibridi prijašnjih algoritama. Postoje različiti tipovi algoritama za sortiranje, ovisno o metodi kojom sortiraju: pomoću raspodjele, odabira, zamjene, ... Neki algoritmi raspodijele originalne podatke u podskupove te onda sortiranje obavljaju nad tim podskupovima — primjer takvog algoritma je *BucketSort*. Hibridni algoritmi sortiraju kombinacijom dva ili više algoritama za sortiranje. Često je slučaj da hibridni algoritmi koriste jedan algoritam za sortiranje kada imamo malo podataka, a neki drugi kada ih imamo puno — primjer takvog algoritma je *Timsort*. Osim duge povijesti algoritama za sortiranje, još jedan razlog zašto ih postoji toliko puno je što ne postoji jedan algoritam koji je najbolji za sve situacije. Različiti algoritmi imaju različite prednosti i mane, pa izbor najboljeg algoritma ovisi o tome koji kriteriji su nam bitni — u nekim slučajevima će nam bitnije od brzine izvršavanja algoritma biti da algoritam ne koristi puno dodatne memorije, stoga bismo se mogli odlučiti za drugačiji algoritam za sortiranje. Također, ako naši podaci nisu potpuno slučajni te znamo neka njihova svojstva, mogli bismo odabrati neki drugi algoritam za sortiranje koji može iskoristiti neke činjenice o tim podacima da bi ih brže sortirao. U teoremu 1.1.1 smo spomenuli da uspoređujući algoritmi ne mogu imati prosječnu vremensku složenost bolju od $O(n \log n)$. No postoje algoritmi koji sortiraju drugim tehnikama, te ako su neki uvjeti za ulazne podatke zadovoljeni, ti algoritmi mogu postići prosječnu vremensku složenost $O(n)$, čime nadmašuju bilo koji algoritam koji samo uspoređuje elemente. No to vrijedi samo u specifičnim slučajevima kada naši podaci imaju neki specijalan oblik (na primjer, troznamenkasti brojevi).

Algoritam	**Izumitelj(i)**	**Godina**
RadixSort	Herman Hollerith	1880.
MergeSort	John von Neumann	1945.
InsertionSort	John Mauchly	1946.
CountingSort	Harold H. Seward	1954.
Digital Sorting		1954.
KeySort		1954.
DistributionSort	Harold H. Seward	1954.
BubbleSort (ExchangeSort)		1956.
Address Calculation Sorting	Isaac, Singleton	1956.

ComparisonSort	E. H. Friend	1956.
Radix List Sort	E. H. Friend	1956.
Two Way InsertionSort	D. J. Wheeler	1957.
RadixSort (modificiran)	P. Hildebrandt, H. Rising, J. Schwartz	1959.
New MergeSort	B. K. Betz, W. C. Carter	1959.
ShellSort	Donald L. Shell	1959.
Cascade MergeSort	R. L. Gilstad	1960.
PolyPhase MergeSort	R. L. Gilstad	1960.
MathSort	W. Feurzeig	1960.
QuickSort	Tony Hoare	1961.
Oscillating MergeSort	Sheldon Sobel	1962.
PatienceSort		1962.
SelectionSort	Kahn	1962.
TopologicalSort		1962.
TournamentSort	K. E. Iverson	1962.
TreeSort	K. E. Iverson	1962.
ShuttleSort		1963.
Biotonic MergeSort	K. Batcher	1964.
HeapSort	J. W. J. Williams	1964.
Theorem H	Douglas H. Hunt	1967.
Batcher Odd-Even MergeSort	K. Batcher	1968.
ListSort	L. J. Woodrum, A. D. Woodall	1969.
Improved QuickSort	Singleton	1969.
Find: The Program	Tony Hoare	1971.
Odd Even Sort	Habermann	1972.
BrickSort	Habermann	1972.
GyratingSort	R. M. Karp	1972.
Binary MergeSort	F. K. Hawang, S. Lin, C. Christen, D. N. Deutsh, G. K. Manacher	1972.
CombSort	Włodzimierz Dobosiewicz	1980.
ProxmapSort	Thomas A. Standish	1980.
SmoothSort	Edsger Dijkstra	1981.
B Sort	Wainright	1985.
UnshuffleSort	Art S. Kagel	1985.
QSort	Wainright	1987.
American Flag Sort		1993.

qSort7	Benteley, McIlory	1993.
New Efficient RadixSort	Arne Anderson, Stefan Nilson	1994.
Self Indexed Sort	Yingxu Wang	1996.
SplaySort	Moggat, Eddy, Petersson	1996.
FlashSort	Karl-Dietrich Neubert	1997.
IntroSort	David Musser	1997.
GnomeSort	Dr. Hamid Sarbazi-Azad	2000.
Timsort	Tim Peters	2002.
SpreadSort	Steven J. Ross	2002.
BeadSort	Joshua J. Arulanandham, Christian S. Claude, Michael J. Dinneen	2002.
BurstSort	Ranja Sinha	2004.
LibrarySort	Michael A. Bender, Martin Farach-Colton, Miguel Mosteiro	2004.
CycleSort	B. K. Haddon	2005.
QuickerSort	R. S. Scowen	2005.
Pancake Sorting	Hal Sudborough	2008.
U Sort	Upendra Singh Aswal	2011.
Counting Position Sort	Nitin Arora	2012.
Novel Sorting Algorithm	R. Shiriniwas, A. Raga Deepthi	2013.
BogoSort		
BucketSort		
J Sort	Jason Morisson	
SS06 Sort	K. K. Sudharajan, S. Chakraborty	
StoogeSort	Howard Fine	
StrandSort		
TrimSort		
Punch Card Sorter	A. S. C. Ross	

Tablica 1.1: Povijest algoritama za sortiranje. Izvor: [3]

RadixSort

Spomenuli smo prije da je većina algoritama koje ćemo opisati u ovom radu uspoređujućeg tipa, tj. za sortiranje koriste samo relaciju „<". *RadixSort* je primjer algoritma koji nije takav. Izumio ga je Herman Hollerith 1880. godine za rad na njegovom elektromehaničkom tabulatoru, a 1954. je na MIT-u Harold H. Seward napisao programski kod za taj algoritam. *RadixSort* je povijesno prvi algoritam za sortiranje. Algoritam sortira brojeve tako da gleda njihove individualne znamenke. Postoje dvije implementacije — jedna koja kreće od najmanje značajne znamenke u broju (LSD — Least Significant Digit), te druga koja kreće od najznačajnije znamenke u broju (MSD — Most Significant Digit). Dakle, *LSD-RadixSort* kreće od najmanje značajne znamenke u broju te sortira ulaz tako da grupira zajedno brojeve koji imaju istu znamenku na toj poziciji — ali unutar tih grupa zadržava originalni poredak elemenata, koristeći neki drugi stabilni algoritam za sortiranje (npr. *BucketSort* ili *CountingSort*). Proces grupiranja se ponavlja za svaku sljedeću znamenku po značajnosti. *MSD-RadixSort* radi na sličan način, ali kreće od najznačajnije znamenke (ako brojevi nemaju jednak broj znamenaka, brojeve koji imaju manje znamenki od broja s najviše znamenki nadopunimo nulama slijeva). Sortira niz elementa na temelju te znamenke, te grupira elemente s istom znamenkom zajedno u zasebnu grupu. Tada rekurzivno opet sortira svaku grupu, krećući od sljedeće najznačajnije znamenke. Na kraju, kada su sve grupe sortirane, algoritam ih spoji u tom poretku.

Stabilnost *LSD-RadixSorta* i *MSD-RadixSorta* ovisi o implementaciji. Za *MSD-RadixSort* postoje implementacije takve da se postiže stabilnost — potrebna nam je dodatna memorija veličine niza koji treba sortirati, no u tom slučaju algoritam više nije *inplace*. Stabilnost LSD-RadixSorta ovisi o stabilnosti pomoćnog algoritma koji se koristi za sortiranje unutar grupa — većina implementacija koristi stabilni algoritam. Ovdje smo pokazali kako *RadixSort* radi na primjeru prirodnih brojeva, no on nije ograničen samo na prirodne brojeve — stringove također možemo reprezentirati prirodnim brojevima. Općenito, bilo koju vrstu podataka možemo sortirati — potrebno je samo da podatke možemo reprezentirati pozicijskom notacijom. *RadixSort* ima vremensku složenost u prosječnom slučaju $O(nk)$, pri čemu je k prosječna duljina ključa.

U tablici 1.2 je prikazan rad *LSD-RadixSorta*.

Početak	**1. prolaz**	**2. prolaz**	**3.prolaz**
326	69**0**	7**0**4	**3**26
453	75**1**	6**0**8	**4**35
608	45**3**	3**2**6	**4**53
835	70**4**	8**3**5	**6**08
751	83**5**	4**3**5	**6**90
435	43**5**	7**5**1	**7**04
704	32**6**	4**5**3	**7**51
690	60**8**	6**9**0	**8**35

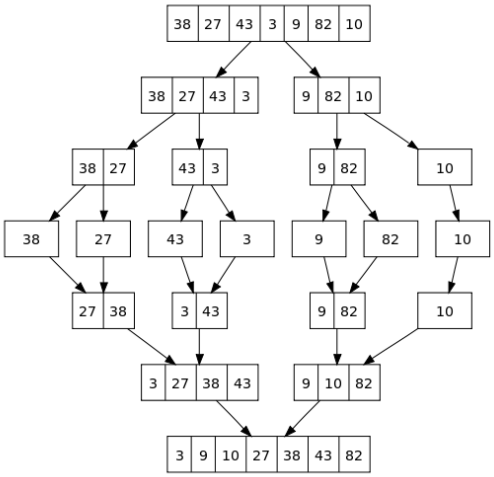
Tablica 1.2: Primjer rada *LSD-RadixSorta*

MergeSort

MergeSort je osmislio John Von Neumann 1945. godine. *MergeSort* je tip algoritma „podijeli pa vladaj". Iako je među najstarijim algoritmima za sortiranje, još uvijek je relevantan za problem sortiranja. Mnogi moderni hibridni algoritmi ga koriste — među njima je i *Timsort*. *MergeSort* radi tako da početni niz podijeli na dvije polovice, rekurzivno pozove samog sebe na svakoj od tih polovica te, nakon što su polovice sortirane, spoji ih u jedan sortiran niz. Niz od n elemenata će se podijeliti u n podnizova od po jednog elementa — u tom trenutku podnizove smatramo trivijalno sortiranima. Zatim ćemo spojiti sortirane podnizove na način da će novi niz koji je nastao kao spoj ta dva niza biti sortiran. To se ponavlja sve dok nam na kraju ne ostane samo jedan niz — to je traženi sortiran niz. Algoritam je efikasan jer iskorištava činjenicu da, kad imamo dva sortirana niza, lako ih je spojiti u jedan sortiran niz. *MergeSort* je među najbržim algoritmima za sortiranje te, pri usporedbi s drugim algoritmima koji brzo sortiraju — *Heapsort* i *Quicksort* — prednost *MergeSorta* je u tome što je stabilan. S druge strane, prednost *Heapsorta* i *Quicksorta* je što su *inplace*, dok najefikasnija implementacija *MergeSorta* nije. Složenost u najboljem, najgorem i prosječnom slučaju je $O(n \log n)$. Vidimo da *MergeSort* postiže donju granicu za složenost uspoređujućih algoritama.

Na slici 1.4 je prikazano kako radi *MergeSort* te njegov pseudokod.

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```



Slika 1.4: *MergeSort* — pseudokod i način rada

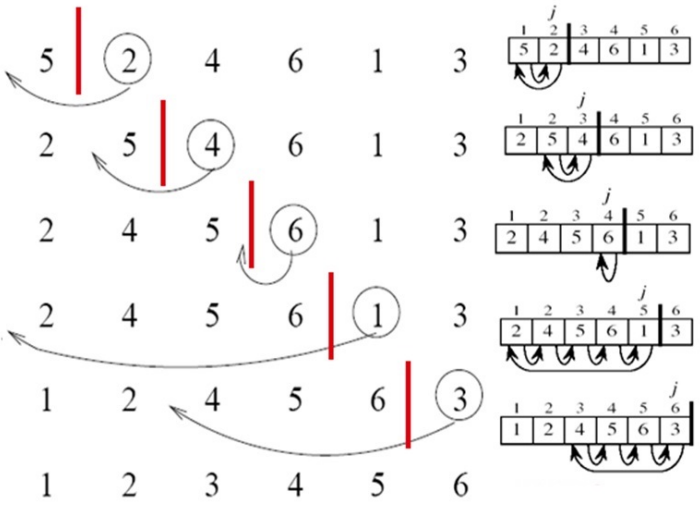
InsertionSort

Ubrzo nakon *MergeSorta*, 1946. godine John Mauchly je osmislio *InsertionSort* — jednostavan, stabilan i prilagođavajući algoritam za sortiranje. *InsertionSort* radi na sličan način kako bi osoba organizirala karte u ruci. Pretpostavimo da su nam dodjeljene karte iste boje i da su na stolu ispred nas, okrenute prema dolje. Ako bismo htjeli imati jednostavan pregled karata, mogli bismo ih sortirati po jačini tako da svaki put kad uzmemo kartu stavimo ju na odgovarajuće mjesto u ruci. Krenuli bismo od prve karte i svaku sljedeću kartu bismo stavili na odgovarajuće mjesto po jačini — to je princip po kojem radi *InsertionSort*. U prvom koraku uzimamo jedan element iz originalnog niza i premještamo ga u (novi) sortiran niz koji je na početku prazan. U svakom sljedećem koraku algoritma uzimamo jedan element iz početnog niza i premještamo ga na odgovarajuće mjesto u sortiranom nizu. U trenutku kada je početni niz prazan, novi niz je sortirana verzija originalnog niza. No jedna mana ovog pristupa je što svaki put kada ubacujemo novi element u sortiran niz, sve veće elemente u sortiranom nizu moramo pomaknuti u memoriji. *InsertionSort* je efikasan za male ili većinom sortirane nizove, ali nije prikladan za veće količine podataka. *ShellSort* je

varijanta *InsertionSorta* koja je efikasnija za veće količine podataka. Zato se *InsertionSort* u današnje vrijeme najčešće koristi kao dio hibridnog algoritma, u kombinaciji s nekim drugim algoritmom koji ima asimptotski dobro ponašanje za veće količine podatka.

Na slici 1.5 je prikazano kako radi *Insertionsort* te njegov pseudokod.

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted
           sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```



Slika 1.5: *InsertionSort* — pseudokod i primjer rada

Poglavlje 2

Timsort

Timsort je osmislio Tim Peters 2002. godine. *Timsort* je stabilan, prilagođavajući, uspoređujući algoritam i ima vremensku složenost u najgorem slučaju $O(n \log n)$ — najbolju moguću za uspoređujuće algoritme. To je hibridni algoritam — kombinacija *MergeSorta* i *InsertionSorta*. Osim Pythona, mnogi drugi jezici koriste *Timsort*: jezici poput Jave, GNU-Octave, ... Najveća prednost *Timsorta* je u tome što zna iskoristiti djelomičnu sortiranost podataka koja se često pojavljuje u stvarnim primjenama. Sortirajući djelomično sortirane podatke *Timsort* puno uštedi na usporedbama, te na poljima koja imaju djelomično sortirane podatke ima odlične performanse.

Ugrubo, *Timsort* radi tako da početno polje podijeli u tokove (vidi sljedeću točku). Svaki put kada identificira sljedeći tok, stavi ga na stog i provjerava određene uvjete za tokove na vrhu stoga. Ako ti uvjeti nisu zadovoljeni — spojimo te tokove i identificiramo sljedeći tok. Ako smo identificirali sve tokove i stog još uvijek ne sadrži samo jedan tok, spojimo sve tokove od vrha stoga prema dnu.

Slijedi opis rada *Timsorta*. Tekst čitavog ovog poglavlja prilagođen je na osnovi dokumenta [10]. Slike koje se koriste u opisu rada preuzete su sa [4].

2.1 Opis rada

Tok

Ulazno polje dijelimo na tokove. *Tok* (engl. *run*) je rastući ili strogo padajući niz elemenata iz početnog polja.

- $a_0 \leq a_1 \leq a_2 \leq \ldots$ je rastući tok;
- $a_0 > a_1 > a_2 > \ldots$ je strogo padajući tok.

U definiciji koristimo strogi uređaj za padajuće tokove jer želimo da algoritam zadrži svojstvo stabilnosti. Algoritam strogo padajući tok pretvori u rastući tako da obrne redoslijed elemenata toka: prvi element polja zamijeni sa zadnjim, drugi s predzadnjim, ... Kad naš tok ne bi bio strogo padajući moglo bi se dogoditi da u njemu postoje jednaki elementi — u tom slučaju, kad bismo ga ovom metodom pretvorili u rastući tok, mogle bi se zamijeniti pozicije jednakih elementa, zbog čega algoritam više ne bi bio stabilan.

Želimo da svi tokovi imaju neku minimalnu duljinu `minrun`. Ako tok ima manje od `minrun` elemenata, koristeći *BinaryInsertionSort* (koji je također stabilan, vidi stranicu 31), nadopunit ćemo tok s toliko elemenata da nakon ubacivanja elemenata tok bude duljine `minrun`. Na taj će način svi tokovi biti duljine barem `minrun`. Očekujemo da za polja koja nemaju pravilnosti nećemo imati duge tokove, pa će s velikom vjerojatnošću svi tokovi biti duljine `minrun`. To je dobro jer znači da su podaci bez pravilnosti skloni savršeno balansiranim spajanjima — u kojima oba toka imaju jednak broj elemenata. U slučaju podataka bez pravilnosti, to je najefikasniji način za spajanje.

Način spajanja tokova

Podaci o toku su sadržani u strukturi `s_slice` — duljina toka i njegova adresa. Tokove spremamo u strukturi `s_MergeState`, u članskoj varijabli `pending` — to je polje strukturâ `s_slice`, koje koristimo kao stog.

Kada identificiramo sljedeći tok, stavljamo ga na vrh stoga `pending`. Ne možemo predugo odgađati spajanje susjednih tokova jer je stog `pending` fiksne duljine, a tokovi koji nisu spojeni zauzimaju više prostora na stogu. Također, ranije spajanje tokova je povoljno jer se onda još uvijek ti tokovi nalaze visoko u memorijskoj hijerarhiji. S druge strane, spajanje želimo odgađati što duže zato da možemo iskoristiti obrasce koji se mogu kasnije pojaviti.

Budući da želimo zadržati svojstvo stabilnosti, možemo spajati samo susjedne tokove na stogu. Naime, pretpostavimo da imamo tri uzastopna toka na vrhu stoga: A, B i C. Nadalje, pretpostavimo da ta tri toka dijele neki zajednički element. Kada bismo spajali tokove A i C, onda barem jedna instanca tog elementa u spojenom toku ne bi više bila u istom relativnom poretku s instancama tog elementa u B. Zato moramo spajati susjedne tokove, tj. A s B ili B sa C. Uzevši sve navedeno u obzir, spajanje se ne obavlja sve dok su zadovoljene sljedeće tri invarijante na četiri najviša elementa na stogu. Označimo ih redom sa Z, A, B, C (pri čemu je C na vrhu), te njihove duljine označimo sa $|Z|$, $|A|$, $|B|$ i $|C|$.

(1) $|A| > |B| + |C|$

(2) $|Z| > |A| + |B|$

(3) $|B| > |C|$

Iz svojstva (3) slijedi da je niz duljina tokova na stogu, čitajući odozgo prema gore, padajući. Iz svojstva (1) i (2) slijedi da, čitajući odozgo prema dolje, duljine tokova rastu brže od Fibonaccijevih brojeva — iz čega slijedi da će duljina stoga biti manja od $\log_\phi n$ (ϕ je omjer zlatnog reza: $\frac{\sqrt{5}+1}{2} \approx 1.618$), tj. relativno mali stog nam je dovoljan za velika polja. Slijedi definicija strategije spajanja tokova.

Definicija 2.1.1. *Optimalni redoslijed spajanja tokova na vrhu stoga je:*

- *Ako nije zadovoljena invarijanta (3) — spoji tokove B i C.*
- *Ako nije zadovoljena invarijanta (1) ili (2)*
  - *ako je* $|A| < |C|$ *— spoji B i A.*
  - *ako je* $|A| \geq |C|$ *— spoji B i C.*

U slučaju kada su A i C jednake duljine, B se spaja s C, jer ako spajamo B i A onda C moramo pomicati za jedno mjesto niže na stogu. Možemo primijetiti da se spajanje odvija samo na prva tri toka na stogu. Sada ćemo dati nekoliko primjera spajanja tokova.
Pretpostavimo da na stogu imamo samo četiri toka, te neka su njihove duljine dane sa $|Z|$, $|A|$, $|B|$, $|C|$, pri čemu najdesniji element predstavlja vrh stoga. Pretpostavimo da je situacija na stogu sljedeća:

$$|Z| = 80, \quad |A| = 30, \quad |B| = 20, \quad |C| = 10\,. \tag{2.1}$$

U ovom slučaju nije zadovoljen uvjet (1) jer $|A| \leq |B| + |C|$, pa s obzirom da je $|C| < |A|$, C se spaja s B. Nakon spajanja vrh stoga izgleda ovako:

$$|Z| = 80, \quad |A| = 30, \quad |BC| = 30\,. \tag{2.2}$$

Ili, pretpostavimo da je sljedeća situacija na vrhu stoga:

$$|Z| = 2000, \quad |A| = 500, \quad |B| = 400, \quad |C| = 1000\,. \tag{2.3}$$

U ovom trenu je bitno napomenuti da algoritam provjerava invarijante (1), (2) i (3) redoslijedom koje su nabrojane, te u skladu s prvom invarijatnom koja je prekršena spaja tokove. U ovom slučaju nisu zadovoljene invarijante (1) i (3), no s obzirom da se prvo provjerava invarijanta (1), spajanje će se izvršiti u skladu s njom. S obzirom da je ovaj put $|A| < |C|$, spojit ćemo tokove A i B. Nakon spajanja, situacija na vrhu stoga je sljedeća:

$$|Z| = 2000, \quad |AB| = 900, \quad |C| = 1000\,. \tag{2.4}$$

U ovom slučaju još uvijek nije zadovoljena invarijanta (3) pa će se spajanje dalje nastaviti. Sukladno definiciji 2.1.1, spojit će se tokovi AB i C, pa vrh stoga izgleda ovako:

$$|Z| = 2000, \quad |ABC| = 1900\,. \tag{2.5}$$

Slično se dogodi i u slučaju (2.2) — invarijanta (3) nije zadovoljena pa se zato spajaju dva najviša toka. Ukratko, spajanje se odvija na sljedeći način: redom se provjeravaju invarijante (1), (2) i (3). Ako je neka invarijanta prekršena, algoritam spoji tokove u skladu s definicijom 2.1.1. Tada imamo novu situaciju na stogu, pa opet krećemo ispočetka provjeravati invarijante (1), (2) i (3). Ovaj proces će se nastavljati dok sve tri invarijante nisu zadovoljene.
U originalnoj implementaciji *TimSorta* nije postojao uvjet (2). Smatralo se da je uvjet (1) na tri najviša toka na stogu dovoljan da se osigura invarijanta da za vrijeme izvršavanja algoritma svaki tok na stogu bude veći od spoja sljedeća dva novija toka na stogu, iz čega bi slijedila posljedica da će duljina stoga biti manja od $\log_\phi n$. No u radu [6] je pokazano da ta invarijanta ne vrijedi. Pretpostavimo da imamo sljedeću situaciju na vrhu stoga:

$$|Y| = 120, \quad |Z| = 80, \quad |A| = 25, \quad |B| = 20, \quad |C| = 30\,. \tag{2.6}$$

Prekršena je invarijanta (1) ($25 < 20 + 30$) i $|A| < |C|$ pa se spajaju A i B.
Zatim je situacija sljedeća:

$$|Y| = 120, \quad |Z| = 80, \quad |AB| = 45, \quad |C| = 30\,. \tag{2.7}$$

Vidimo da vrijedi $120 \leq 80 + 45$, tj. nije zadovoljen uvjet

$$|T_i| > |T_{i+1}| + |T_{i+2}| \tag{2.8}$$

za sve $i \in \{0, \ldots, n-3\}$, gdje je n duljina stoga (T_i označava i-ti tok na stogu: T_0 je na dnu stoga). Ovaj uvjet ćemo zvati *Fibonaccijevo svojstvo*. S obzirom da Fibonaccijevo svojstvo nije zadovoljeno, ne možemo zaključiti da će duljina stoga biti manja od $\log_\phi n$. No u [5] su dokazali da uvođenjem (2) je svojstvo (2.8) zadovoljeno. Problem nastaje u tome što je stog fiksne duljine — 85 — što bi u slučaju da vrijedi svojstvo (2.8) bilo dovoljno za 2^{64} podataka. Ali s obzirom (2.8) nije zadovoljeno, duljine tokova ne rastu dovoljno brzo, te je stog duljine 85 dovoljan za 2^{49} podataka — što je još uvijek previše podataka za današnja računala da bi se mogla izazvati greška u izvršavanju.

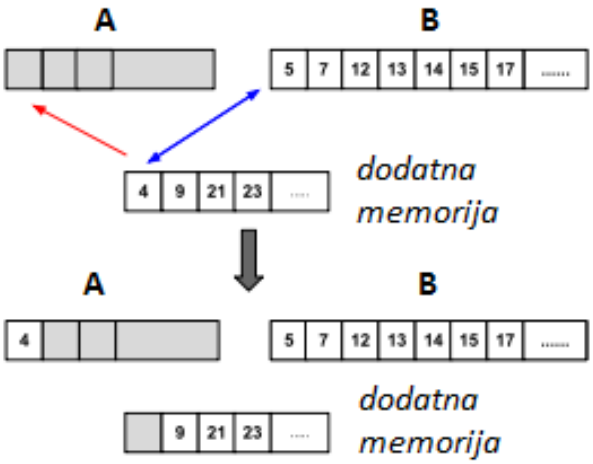
Cilj ovih invarijanti je da pokušaju balansirati duljine tokova na stogu, a da ne moraju pamtiti velik broj tokova. Ovo je efikasno za podatke bez pravilnosti jer za njih je vjerojatno da će svi tokovi biti duljine `minrun`, te će se na kraju napraviti savršeno balansirano spajanje. Na primjer, pretpostavimo da ne koristimo vrijednost `minrun`, tj. ne postoji donja granica na duljinu toka, te da imamo tokove duljina 128, 64, 32, 18, 4, 2 i 2. Invarijante (1), (2) i (3) bi bile zadovoljene sve dok posljednji tok duljine 2 ne bi došao na stog. U tom trenu bi počelo spajanje (nije zadovoljena invarijanta (1)) koje bi se nastavilo sve do zadnjeg toka, tj. imali bismo sedam spajanja na tokovima jednakih duljina, što je najefikasnije za podatke bez pravilnosti.

S druge strane, algoritam dobro sortira djelomično sortirane podatke upravo zbog tokova koji imaju jako različite duljine, što ćemo uskoro pokazati.

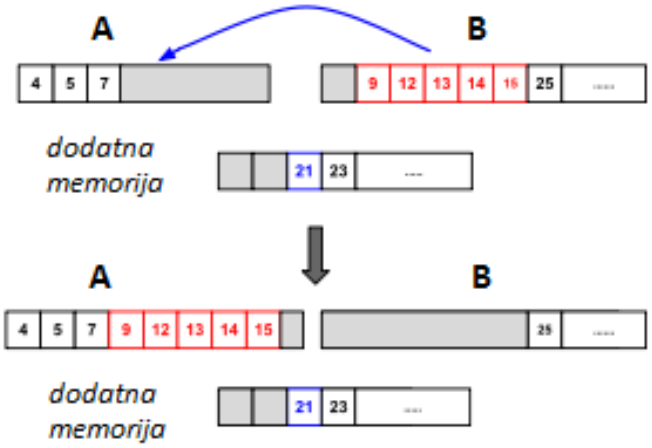
Dodatna memorija za spajanje

Iako postoje načini da tokove A i B spojimo tako da algoritam bude *inplace*, u praksi bi to bilo presporo. Iz tog razloga, kad spajamo tokove duljina $|A|$ i $|B|$, koristimo dodatnu memoriju veličine $\min(|A|, |B|)$. Slijedi detaljni opis spajanja.

Ako je $|A| \leq |B|$, spajanje obavlja funkcija `merge_lo` (vidi stranicu 38). Ona prvo kopira A u dodatnu memoriju, te zatim spaja elemente s lijeva na desno iz dodatne memorije i elemente iz B te ih sprema na početak memorije gdje je A bio zapisan (slika 2.1). Jednom kad je `merge_lo` završio, sortirano polje će biti spremljeno u zajedničkoj memoriji od A i B (slika 2.3).



Slika 2.1: Sortirani elementi se spremaju u originalnu lokaciju toka A



Slika 2.2: Premještanje dijela toka za vrijeme galopiranja

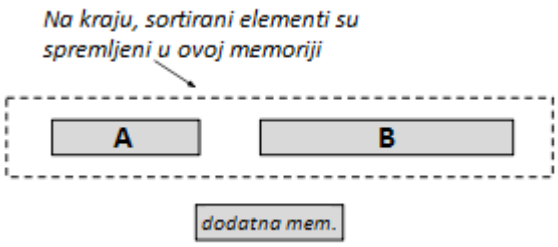
Na slikama 2.1 i 2.2, A i B označavaju mjesta u memoriji gdje su originalno bila spremljena ta polja. Nakon što smo odredili da je A manje polje, kopirali smo ga u dodatnu memoriju te sortirane elemente zapisujemo na početak memorije gdje je A bio spremljen. U nastavku teksta, $A[0]$ će nam uvijek označavati prvi element u dodatnoj memoriji. Znači, na slici 2.1, prije premještanja elementa 4, $A[0] = 4$. Nakon što smo iz dodatne memorije premjestili 4, bit će $A[0] = 9$. Analogno vrijedi za B — $B[0]$ je prvi element iz polja B koji još nije prebačen u zajedničku memoriju. Na slici 2.3 je prikazana zajednička memorija od A i B u kojoj su na kraju spremljeni sortirani podaci — to ćemo zvati *MergeMemory*.

Jedna od stvari na koje treba paziti je da se usred spajanja može dogoditi iznimka. U tom slučaju preostale elemente iz dodatne memorije treba kopirati natrag u slobodno mjesto u zajedničkoj memoriji od A i B, zato da slučajno sortirano polje ne bi sadržavalo duplikate iz B. No implementirati to nije teško, jer istu stvar moramo napraviti ako dođemo do kraja polja B prije nego A, tako da možemo iskoristiti taj dio koda. U slučaju da je $|B| < |A|$, spajamo polja pomoću funkcije `merge_hi` na isti način uz malu promjenu: kopiramo B u dodatnu memoriju i spojimo tokove tako da zapisujemo sortirane elemente počevši od kraja originalnoga polja gdje se B nalazio, znači zdesna na lijevo. Još jedno poboljšanje koje možemo napraviti je da prije nego što počnemo spajati susjedne tokove A i B, prvo galopiramo (vidi stranicu 35) da nađemo na kojem mjestu u A bi se trebao nalaziti element $B[0]$. Elementi od A koji su manji od $B[0]$ se već nalaze na svojoj konačnoj poziciji u *MergeMemory* te kad spajamo A i B možemo zanemariti taj dio od A. Na isti način u B tražimo gdje bi se trebao nalaziti $A[-1]$ — zadnji element od A — te elemente u B koji dolaze poslije njega možemo ignorirati. Na taj smo način smanjili veličinu potrebne dodatne memorije za količinu elemenata koje smo izbacili iz razmatranja. U slučaju

kada nema pravilnosti u našim podacima, ova pretraživanja prije spajanja se ne isplate — u prosjeku će algoritam raditi više posla. No kada radimo s podacima koji skrivaju neku strukturu, ova pretraživanja se mogu jako isplatiti s obzirom na potrebnu količinu memorije, broj usporedbi i kopiranja. Ovo je jedan primjer dodatnog troška pri spajanju tokova koji prihvaćamo radi efikasnosti u slučaju strukturiranih podataka.

Algoritam za spajanje tokova

U ovom odjeljku ćemo objasniti kako točno funkcija `merge_lo` radi (`merge_hi` radi analogno). Spajanje radimo tako da usporedimo prvi element od A s prvim elementom od B. Ako je $B[0]$ manji, njega premještamo u *MergeMemory* — inače premještamo $A[0]$. Pretpostavimo da je u ovom slučaju $A[0]$ bio manji. Nakon što smo premjestili $A[0]$, sljedeći element u dodatnoj memoriji postaje $A[0]$. Znači, opet usporedimo $A[0]$ s $B[0]$ te opet manji od njih premjestimo na sljedeće slobodno mjesto u *MergeMemory*, te nastavimo dalje — uspoređujemo sljedeći element iz polja u kojem je bio manji element s elementom iz drugog polja koji je bio veći u prošloj usporedbi — tako uspoređujemo elemente „jedan po jedan" sve do kraja. Na isti način *MergeSort* spaja sortirana polja. Uz to pratimo koliko puta za redom je element iz istog toka bio manji. Ako taj broj dostigne vrijednost `min_gallop` prelazimo u *galopirajući način rada* (vidi stranicu 35) — tražimo prvi element iz jednog polja u drugom tako da uspoređujemo taj element s elementima u drugom polju na pozicijama $2^k - 1$. Na taj način znamo u kojem rasponu elemenata treba ležati naš element. Na primjer, mogli bismo tražiti gdje u B pripada element $A[0]$. Kada nađemo to mjesto, sve elemente prije toga mjesta možemo prebaciti u *MergeMemory* (slika 2.2), te još $A[0]$ neposredno nakon njih.



Slika 2.3: *MergeMemory*

Zatim pretražimo A da vidimo gdje $B[0]$ pripada i prebacimo sve elemente prije tog mjesta u *MergeMemory*, te nakon njih $B[0]$. Onda opet tražimo $A[0]$ u $B, \ldots$ To se ponav-

lja sve dok barem jedno od tih pretraživanja nađe duljinu dijela toka za premjestiti jednaku ili veću od `min_gallop`. U trenutku kada taj uvjet nije zadovoljen vraćamo se na standardno uspoređivanje elemenata „jedan po jedan". Struktura `s_MergeState` sadrži vrijednost `min_gallop` koju inicijaliziramo na `MIN_GALLOP`, a `merge_lo` i `merge_hi` podešavaju `min_gallop` ovisno o tome da li se galopiranje isplatilo. Kada se isplati (tj. našli smo barem `min_gallop` podataka za kopirati) smanjimo `min_gallop`, inače ga povećamo — na taj način olakšavamo, odnosno otežavamo ponovno galopiranje. U nastavku ćemo to detaljnije objasniti.

Galopiranje

Pretpostavimo da je A kraći tok. Kada galopiramo, prvo tražimo poziciju od $A[0]$ u B tako da $A[0]$ uspoređujemo redom s $B[0]$, $B[1]$, $B[3]$, $B[7], \ldots B[2^j - 1], \ldots$ sve dok ne nađemo k tako da vrijedi $B[2^{k-1}] < A[0] \leq B[2^k - 1]$. Za to treba najviše $\log_2 |B|$ usporedbi. Nakon što smo našli takav k, naš $A[0]$ se nalazi u rasponu elemenata duljine $2^{k-1} - 1$, te binarnim pretraživanjem unutar tog raspona nakon $k - 1$ usporedbi nađemo točno na kojem mjestu se treba nalaziti $A[0]$. Zatim kopiramo sve elemente iz B prije tog mjesta u *MergeMemory* te nakon toga kopiramo još $A[0]$. Ovom kombinacijom traženja u najgorem slučaju nećemo premašiti $2\log_2(|B|)$ usporedbi. Binarnim pretraživanjem, to mjesto bismo mogli naći s $\lceil \log_2(|B| + 1) \rceil$ usporedbi, no binarno pretraživanje će uvijek napraviti toliko koraka, bez obzira na to na kojoj poziciji se $A[0]$ treba nalaziti — to znači da će binarno pretraživanje napraviti više koraka u skoro svim slučajevima — osim kada je tok jako dugačak. Ako podaci nemaju pravilnosti i tokovi su jednake duljine, vjerojatnost da $A[0]$ pripada na poziciju od $B[0]$ je $1/2$, na poziciju od $B[1]$ je $1/4, \ldots$ Općenito, vjerojatnost da za neki k $A[0]$ pripada na poziciju $B[k]$ je $1/2^{k+1}$, što znači da postoji mala vjerojatnost da će za podatke bez pravilnosti tokovi biti dugački. Zato je galopiranje bolje od običnog binarnog pretraživanja. Galopiranje je bolje i od linearnog pretraživanja. Ako podaci imaju strukturu ili sadržavaju puno duplikata, velika je vjerojatnost da će se pojaviti dugački tokovi, a u tom slučaju smo smanjili broj usporedbi s $O(|B|)$ na $O(\log |B|)$ što je značajno ubrzanje.

Galopiranje se ne isplati uvijek — u nekim slučajevima će trebati više vremena od linearnog pretraživanja. Jedan od razloga je što u nekim slučajevima galopiranje i linearno pretraživanje trebaju jednak broj usporedbi da odrede na kojoj poziciji pripada element, a pozivanje još jedne funkcije (galopiranje) zahtjeva dodatne resurse — za svaku usporedbu pozivati još jednu funkciju bi bilo preskupo. Nadalje, u određenim slučajevima galopiranje će trebati više usporedbi od običnog linearnog pretraživanja. Promotrimo tablicu 2.1.

Ako se $A[0]$ treba nalaziti prije $B[0]$, linearno pretraživanje i galopiranje trebaju samo jednu usporedbu da to ustanove, ali pozivanje funkcije galopiranja je skuplje. Znači u ovom slučaju je isplativije koristiti linearno pretraživanje. Ako se $A[0]$ nalazi prije $B[1]$

imamo istu situaciju: oboje trebaju dvije usporedbe da to ustanove, stoga je opet linearno pretraživanje efikasnije. Ako $A[0]$ pripada neposredno prije B[3], galopiranje treba četiri usporedbe: tri usporedbe mu trebaju da ustanovi $A[0] \leq B[3]$, ali mu je potrebna još jedna usporedba jer nakon tri usporedbe galopiranja znamo da $A[0] > B[1]$ i $A[0] \leq B[3]$, što znači da $A[0]$ pripada na mjesto od $B[2]$ ili $B[3]$. U ovom slučaju linearno pretraživanje također zahtjeva četiri usporedbe, ali u slučaju da $A[0]$ pripada na mjesto od $B[2]$, linearno pretraživanje će trebati tri usporedbe, a galopiranje još uvijek četiri. Iako se to ne čini kao velika razlika, to je povećanje broja usporedbi za 33%, što nije zanemariv gubitak.

Iz tablice 2.1 vidimo da galopiranje ne radi manje usporedbi od linearnog pretraživanja sve do indeksa $i = 6$. Dapače, kao što smo pokazali gore, za indekse $i = 4$ ili $i = 2$ galopiranju je potrebno više usporedbi, a u ostalim slučajevima jednak broj usporedbi. Od indeksa $i = 6$ nadalje galopiranje uvijek treba manje usporedbi — to znači da nam se galopiranje isplati jedino kada se inicijalni element ne nalazi među prvih 6 elemenata. Znači ne želimo odmah početi galopirati, nego tek kada nam podaci pokažu naznaku da bi moglo biti strukture u podacima. Iz toga razloga za inicijalnu vrijednost uzimamo `MIN_GALLOP` = 7.

Indeks u B gdje A[0] pripada	Linearno pretraživanje	Galopiranje	Binarno pretraživanje	Galopiranje + Binarno pretraživanje
0	1	1	0	2
1	2	2	0	2
2	3	3	1	4
3	4	3	1	4
4	5	4	2	6
5	6	4	2	6
6	7	4	2	6
7	8	4	2	6
8	9	5	3	8
9	10	5	3	8
10	11	5	3	8
11	12	5	3	8

Tablica 2.1: Broj usporedbi za linearno pretraživanje i galopiranje

Ovisno o strukturi podataka koje sortiramo, prag za galopiranje — `min_gallop` — može biti previsok ili prenizak. Nekada ćemo krenuti galopirati i kada su naši podaci bez strukture, tj. nema neke pravilnosti koju bismo mogli iskoristiti te je zato vrlo vjerojatno da ćemo brzo izaći iz galopiranja. S druge strane (vidi [10]), za neke podatke se uvijek isplati galopirati. Zato u strukturi `merge_struct` čuvamo vrijednost `min_gallop`, a u

funkcijama `gallop_left` i `gallop_right` tu vrijednost prilagođavamo. Svaki put kad se galopiranje isplatilo, smanjimo vrijednost `min_gallop` za jedan, tako da je lakše opet ući u galopiranje. Ako se nije isplatilo, tj. ako smo u galopirajućem načinu rada kopirali manje od `min_gallop` podataka, prestajemo galopirati i povećamo vrijednost `min_gallop` za jedan. Na taj način se trudimo da ne galopiramo često ako nema strukture u podacima, a češće galopiramo kada je ima. U slučaju kada podaci nemaju pravilnosti, `min_gallop` će se izrazito povećati pa će biti puno teže opet početi galopirati. Ako podaci imaju strukturu, svaki put kad uspješno galopiramo `min_gallop` će se smanjiti te će u trenutnom spajanju biti lakše nastaviti galopirati, a budućim spajanjima lakše započeti galopirati. Ukratko, galopiranje je efikasno jer u slučaju da se ne isplati bit će puno teže opet ući u galopiranje, a ako se isplati bit će lakše ući i uštedjet ćemo puno na broju usporedbi.

S ovim smo opisali kako galopiranje radi u funkciji `merge_lo`. Funkcija `merge_hi` radi analogno uz malu promjenu: `merge_hi` spaja od najvećeg prema najmanjem elementu, tj. s desna na lijevo, pa trebamo u tom smjeru i galopirati — znači galopiranje umjesto na prvom počinje na zadnjem elementu polja. Iz tog razloga funkcije `gallop_left` i `gallop_right` imaju argument koji određuje od kojeg mjesta u polju treba krenuti galopiranje. Stoga, moguće je krenuti galopirati od bilo koje pozicije u polju tako da nastavimo gledati elemente od te pozicije s pomacima od 2^k za $k \geq 0$, no u originalnom kodu funkcije se pozivaju s argumentom 0 ili $n - 1$, pri čemu je n broj elemenata u toku. Iako bi se teoretski moglo pokušati iskoristiti da galopiranje počinje i na drugim pozicijama osim prve i zadnje, nije jasno kako bi se to moglo efikasno primijeniti.

`minrun`

Ovdje opisujemo način izračunavanja parametra `minrun`. Neka je N veličina polja. Ako je $N < 64$ onda je `minrun` $= N$, tj. s obzirom da je polje dovoljno malo, cijelo polje sortiramo *BinaryInsertionSortom* (vidi stranicu 31) — zbog dodatnih troškova ne isplati se ništa kompliciranije.

Kao što smo spomenuli, za podatke bez pravilnosti najefikasnije spajanje je kada su tokovi savršeno balansirani. Zato kada je N potencija od 2 želimo da naša vrijednost `minrun` također bude potencija od 2 (vidi stranicu 47). Pokazalo se da su 16, 32, 64 i 128 dobre opcije — za manje vrijednosti je previše poziva funkcija, a za veće vrijednosti količina premještanja podataka u *BinaryInsertionSortu* je prevelika. Zato, kada je N potencija od 2, biramo 32 kao vrijednost `minruna`.

No 32 ili neka druga potencija od 2 nije dobra vrijednost u ostalim slučajevima — kada N *nije* potencija od 2. Pretpostavimo da je $N = 2112$ i da smo uzeli `minrun` vrijednost 32. Ako su naši podaci bez pravilnosti, vjerojatno je da će se podijeliti u 66 tokova, svaki duljine 32. Spajanje prva 64 toka biti će savršeno balansirano — sva spajanja će biti nad tokovima jednakih duljina — i kao rezultat će imati jedan tok duljine 2048. Na kraju će

nam na stogu ostati tokovi duljine 2048 i 64 koje treba spojiti — tj. morat ćemo potrošiti puno usporedbi samo da bismo 64 podatka doveli na odgovarajuće mjesto. Ali, kada bi za podatke bez pravilnosti odabrali 33 kao vrijednost `minruna`, podaci bi se podijelili u 64 toka, svaki duljine 33. Tada bi sva spajanja bila savršeno balansirana.

Znači, ako zapišemo N kao $N = q \cdot \texttt{minrun} + r$, želimo izbjeći slučaj u kojem je q malo veći od potencije od 2, bez obzira na ostatak r (gornji slučaj) ili slučaj u kojemu je q potencija od 2 i $r > 0$. Tada se dogodi slična situacija kao gornja — zadnje spajanje dovodi samo r brojeva na njihove pozicije, što je puno manje od N. Radi ovih razloga `minrun` biramo iz intervala $[32, 64]$ tako da je q potencija od 2 ili ako to nije moguće je blizu, ali strogo manje od potencije od 2. Za računanje `minruna` je zadužena funkcija `merge_compute_minrun`, koja će biti objašnjena u sljedećem poglavlju.

2.2 Kod

U ovom odjeljku prolazimo kroz implementaciju *Timsorta*. Prvo prolazimo kroz pomoćne funkcije i strukture te na kraju kroz sam algoritam koji sortira — `list_sort_impl`. Sav kod je preuzet s [11].

Osnovna struktura i pomoćne funkcije

```
typedef struct {
    PyObject **keys;
    PyObject **values;
} sortslice;
```

`sortslice` nam predstavlja osnovnu strukturu s kojom radimo. Sadrži pokazivače na polje koje trebamo sortirati: pokazivač na polje ključeva `keys` i pokazivač na polje odgovarajućih vrijednosti `values`, tj. `keys[i]` ima njemu odgovarajuću vrijednost `values[i]`. Ako je `values==NULL`, onda su ključevi ujedno i vrijednosti.

Sada navodimo pomoćne funkcije koje nam služe da se ključevi i vrijednosti uvijek zajedno premještaju i kopiraju.

```
Py_LOCAL_INLINE(void)
sortslice_copy(sortslice *s1, Py_ssize_t i,
               sortslice *s2, Py_ssize_t j)
{
    s1->keys[i] = s2->keys[j];
    if (s1->values != NULL)
        s1->values[i] = s2->values[j];
}


Py_LOCAL_INLINE(void)
sortslice_copy_incr(sortslice *dst, sortslice *src)
{
    *dst->keys++ = *src->keys++;
    if (dst->values != NULL)
        *dst->values++ = *src->values++;
}
```

```
Py_LOCAL_INLINE(void)
sortslice_copy_decr(sortslice *dst, sortslice *src)
{
    *dst->keys-- = *src->keys--;
    if (dst->values != NULL)
        *dst->values-- = *src->values--;
}


Py_LOCAL_INLINE(void)
sortslice_memcpy(sortslice *s1, Py_ssize_t i,
                 sortslice *s2, Py_ssize_t j, Py_ssize_t n)
{
    memcpy(&s1->keys[i], &s2->keys[j], sizeof(PyObject *) * n);
    if (s1->values != NULL)
     memcpy(&s1->values[i], &s2->values[j], sizeof(PyObject *) * n);
}


Py_LOCAL_INLINE(void)
sortslice_memmove(sortslice *s1, Py_ssize_t i,
                  sortslice *s2, Py_ssize_t j, Py_ssize_t n)
{
    memmove(&s1->keys[i], &s2->keys[j], sizeof(PyObject *) * n);
    if (s1->values != NULL)
     memmove(&s1->values[i], &s2->values[j], sizeof(PyObject *) * n);
}


Py_LOCAL_INLINE(void)
sortslice_advance(sortslice *slice, Py_ssize_t n)
{
    slice->keys += n;
    if (slice->values != NULL)
        slice->values += n;
}
```

Uvodimo i dva pomoćna makroa koja služe uspoređivanju:

```
#define ISLT(X, Y) (*(ms->key_compare))(X, Y, ms)
```

Uspoređuje x i y funkcijom `ms->key_compare` (vidi dolje). Povratna vrijednost je -1 ako je došlo do greške, 1 ako je $x < y$ i 0 ako je $x \geq y$.

```
#define IFLT(X, Y) if ((k = ISLT(X, Y)) < 0) goto fail;  \
           if (k)
```

Uspoređuje x i y koristeći „<”. Ako je došlo do greške, izvrši se naredba `goto fail`. Inače se započne „`if(k)`” blok, koji se izvrši ako je $x < y$.

Pomoćne strukture i funkcije

Definiramo u kodu konstante:

```
#define MAX_MERGE_PENDING 85
```

Veličina stoga na koji spremamo tokove — S obzirom da će za n podataka visina stoga biti manja od $\log_\phi n$, možemo sortirati polja veličine do $32\phi^{\texttt{MAX_MERGE_PENDING}}$.
`MAX_MERGE_PENDING` = 85 je dovoljno veliko za polja veličine 2^{64} — previše podataka za bilo koje današnje računalo.

```
#define MIN_GALLOP 7
```

`min_gallop` se inicijalizira na `MIN_GALLOP`. Jednom kad krenemo galopirati, nastavljamo sve dok oba toka ne nađu manje od `min_gallop` elemenata za kopiranje.

```
#define MERGESTATE_TEMP_SIZE 256
```

U `s_MergeState` (definirano dolje) definiramo polje `temparray` veličine `MERGESTATE_TEMP_SIZE` koje nam služi kao privremeno pomoćno polje pri spajanju dva toka. Na taj način možemo za tokove koji su dovoljno mali izbjeći dodatnu alokaciju memorije.

Definiramo u kodu strukture:

- `s_slice` — struktura koja sadrži podatke o toku.

```
struct s_slice {
    sortslice base; /* adresa početka toka */
    Py_ssize_t len; /* duljina toka */
};
```

- `s_MergeState` — struktura koja sadrži stog `pending` na koji spremamo tokove i dodatnu memoriju `a` koju koristimo pri spajanju dva toka.

```
typedef struct s_MergeState MergeState;
struct s_MergeState {

    /* Varijabla koja kontrolira kad počinjemo galopirati */
     * inicijalizira se na MIN_GALLOP. Merge_lo i Merge_hi */
     * je smanjuju za 1 ako se galopiranje isplatilo, */
     * inače povećavaju za 1.
     */
    Py_ssize_t min_gallop;

    /* a je dodatna memorija koju koristimo pri
     * spajanju dva toka. U nju stane alloced podatka
     */
    sortslice a;
    Py_ssize_t alloced;

    /* pending je stog od n tokova */
    int n;
    struct s_slice pending[MAX_MERGE_PENDING];

    /* ako je potrebna privremena memorija dovoljno mala,
     * a pokazuje na temparray - u tom slučaju ne
     * trebamo alocirati novu memoriju
     */

    PyObject *temparray[MERGESTATE_TEMP_SIZE];

    /* funkcije koje koristimo za uspoređivanje */

    int (*key_compare)(PyObject *, PyObject *, MergeState *);

    PyObject *(*key_richcompare)(PyObject *, PyObject *, int);

    int (*tuple_elem_compare)(PyObject *, PyObject *,
                              MergeState *);
} MergeState;
```

merge_init, merge_freemem i merge_getmem su pomoćne funkcije za alokaciju dodatne memorije. merge_freemem oslobađa privremenu memoriju a u strukturi s_MergeState, merge_init je konstruktor za s_MergeState, a merge_getmem alocira potrebnu memoriju za sljedeći tok ako on ne stane u dodatnu memoriju a.

Funkcija MERGE_GETMEM alocira memoriju za tok. Ako tok stane u a vrati 0 — nije potrebno opet alocirati memoriju. Inače, alocira potrebnu memoriju.

```
#define MERGE_GETMEM(MS, NEED) ((NEED) <= (MS)->alloced ? 0 :   \
                                merge_getmem(MS, NEED))
```

binarysort

Funkcija binarysort je *BinaryInsertionSort* — prima dio polja i sortira ga. Radi na isti način kao i InsertionSort — jedina razlika je da kad ubacujemo novi element u sortirani dio polja, to ne radimo linearnim nego binarnim pretraživanjem. Kao argumente prima lo, hi (početak, odnosno prvi element izvan polja koje želimo sortirati) i start — dio polja $[\texttt{lo}, \texttt{start}\rangle$ je već sortiran. Funkcija u slučaju greške vraća -1, inače vraća 0.

```
static int
binarysort(MergeState *ms, sortslice lo,
           PyObject **hi, PyObject **start)
{
    Py_ssize_t k;

    /* l=left, r=right -> rubovi za binarno pretraživanje */
    PyObject **l, **p, **r;
    PyObject *pivot;

    assert(lo <= start && start <= hi);

    /* početak od jednog elementa je uvijek sortiran */
    if (lo.keys == start)
        ++start;

    /* treba sortirati dio [start,hi> */
    for (; start < hi; ++start) {
```

```
        l = lo.keys;    /* granice sortiranog polja [lo,start> */
        r = start;
        pivot = *r;     /* vrijednost koju ubacujemo u polje */

        assert(l < r);

        /* binarno pretraživanje s granicama l i r */
        do {
            p = l + ((r - l) >> 1);
            IFLT(pivot, *p)
                r = p;
            else
                l = p+1;
        } while (l < r);
        assert(l == r);          /* pivot pripada na poziciju l */

        /* elemente između pozicije l i start pomičemo za jedan */
        for (p = start; p > l; --p)
            *p = *(p-1);
        *l = pivot;

        /* isto to napravimo za vrijednosti */
        if (lo.values != NULL) {
            Py_ssize_t offset = lo.values - lo.keys;
            p = start + offset;
            pivot = *p;
            l += offset;
            for (p = start + offset; p > l; --p)
                *p = *(p-1);
            *l = pivot;
        }
    }
    return 0;

 fail:
    return -1;
}
```

count_run i reverse_slice

Funkcija count_run identificira tokove. Kao argumente prima lo i hi — lo je pozicija prvog elementa u početnom polju koji nije dio nekog toka a hi − 1 je pozicija zadnjeg elementa početnog polja. Funkcija kreće od pozicije lo i traži kraj trenutnog toka. Kao povratnu vrijednost vraća duljinu identificiranog toka, a u slučaju greške vraća −1.

```
static Py_ssize_t
count_run(MergeState *ms, PyObject **lo,
          PyObject **hi, int *descending)
{
    Py_ssize_t k;
    Py_ssize_t n;

    assert(lo < hi);

    /* descending označava je li tok rastući (0) */
    /* ili strogo padajući (1) */
    *descending = 0;
    ++lo;           /* sad lo pokazuje na 2. element u toku */
    if (lo == hi)
        return 1;

    n = 2;
    IFLT(*lo, *(lo-1)) {    /* ->2. element je manji od 1. */
        *descending = 1;    /* -> tok je strogo padajući */

        /* koliko je brojeva s početka niza u */
        /* strogo padajućem poretku */
        for (lo = lo+1; lo < hi; ++lo, ++n) {
            IFLT(*lo, *(lo-1))
                ;
            else
                break;
        }
    }
    else {          /* 2. element je >= od 1. -> tok je rastući */

        /* koliko je brojeva s početka niza u */
        /* rastućem poretku */
```

```
        for (lo = lo+1; lo < hi; ++lo, ++n) {
            IFLT(*lo, *(lo-1))
                break;
        }
    }

    return n;
fail:
    return -1;
}
```

Funkcija `reverse_slice` strogo padajuće tokove pretvara u rastuće. Kao argumente prima `lo` i `hi` — `lo` je pozicija prvog elementa u toku, a `hi` − 1 je pozicija zadnjeg elementa toka. Funkcija radi tako da zamjenjuje elemente padajućeg toka na različitim krajevima — prvi element sa zadnjim, drugi s predzadnjim, ... Nema povratne vrijednosti. Funkcija `reverse_sortslice` radi istu stvar, samo osigura da funkcija `reverse_slice` napravi promjenu na ključevima i vrijednostima.

```
static void
reverse_slice(PyObject **lo, PyObject **hi)
{
    assert(lo && hi);

    --hi;
    while (lo < hi) {
        PyObject *t = *lo;
        *lo = *hi;
        *hi = t;
        ++lo;
        --hi;
    }
}

static void
reverse_sortslice(sortslice *s, Py_ssize_t n)
{
    reverse_slice(s->keys, &s->keys[n]);
    if (s->values != NULL)
        reverse_slice(s->values, &s->values[n]);
}
```

gallop_left i gallop_right

Sada ćemo objasniti kako radi funkcija `gallop_left`. Funkcija kao argumente prima element `key`, sortirano polje `a`, veličinu polja `a` — `n` i cjelobrojni pomak `hint`. Povratna vrijednost funkcije je cijeli broj `k` tako da vrijedi: $\texttt{a}[\texttt{k}-1] < \texttt{key} \leq \texttt{a}[\texttt{k}]$. Znači, funkcija traži poziciju od `key` u polju `a` na kojoj `key` pripada. Ako polje `a` sadrži elemente koji su jednaki `key`, vratit će poziciju lijevo od najljevijeg jednakog elementa.

Funkcija radi tako da redom uspoređuje `key` s elementima polja `a`, prvo s $\texttt{a}[\texttt{hint}]$ te onda s $\texttt{a}[\texttt{hint} \pm 2^i]$ za $i = 0, 1, 2, \ldots$ Pointer na `a` smo pomaknuli za `hint` (linija 9) — tj. `*a` je element polja `a` na mjestu `hint`. Znači, prvo uspoređujemo `key` s $\texttt{a}[\texttt{hint}]$ (linija 15). Ako je $\texttt{a}[\texttt{hint}] < \texttt{key}$, znači da se `key` nalazi desno od $\texttt{a}[\texttt{hint}]$ pa galopiramo udesno (linije 17–33). Varijabla `ofs` (*offset*, tj. pomak) redom poprima vrijednosti $2^i, i = 0, 1, 2, \ldots$ Dok god je $\texttt{ofs} < \texttt{maxofs}$ uspoređujemo $\texttt{a}[\texttt{ofs}]$ i `key`, pri čemu je $\texttt{maxofs} = n - \texttt{hint}$, tj. duljina polja `a` krećući od $\texttt{a}[\texttt{hint}]$.

U slučaju da je u 15. liniji $\texttt{key} \leq \texttt{a}[\texttt{hint}]$, galopiramo ulijevo (linije 34–56). Sve je isto kao kad galopiramo udesno, samo što sad umjesto dodavanja `ofs` oduzimamo ga od pozicije $\texttt{a}[\texttt{hint}]$. Nakon što nađemo poziciju tako da vrijedi $\texttt{a}[\texttt{hint} - \texttt{ofs}] < \texttt{key} \leq \texttt{a}[\texttt{hint} - \texttt{lastofs}]$, binarnim pretraživanjem nađemo točno mjesto unutar intervala.

```
static Py_ssize_t
gallop_left(MergeState *ms, PyObject *key, PyObject **a,
            Py_ssize_t n, Py_ssize_t hint)
{
1  Py_ssize_t ofs;
2  Py_ssize_t lastofs;
3  Py_ssize_t k;
4
5
6   assert(key && a && n > 0 && hint >= 0 && hint < n);
7
8
9    a += hint;
10   lastofs = 0;
11   ofs = 1;
12
13    /* a[hint] < key  -> galopiramo udesno dok ne vrijedi */
14    /* a[hint + lastofs] < key <= a[hint+ofs] */
```

```
15   IFLT(*a, key) {
16      const Py_ssize_t maxofs = n - hint;
17      while (ofs < maxofs) {
18           IFLT(a[ofs], key) {
19               lastofs = ofs;
20               ofs = (ofs << 1) + 1;
21               if (ofs <= 0)          /* integer overflow */
22                   ofs = maxofs;
23           }
24           else    /* key <= a[hint + ofs] */
25               break;
26       }
27       if (ofs > maxofs)
28           ofs = maxofs;
29
30      /* Podesi offsetove da budu u odnosu na &a[0]. */
31       lastofs += hint;
32       ofs += hint;
33   }
34   else {
35
36       /* key <= a[hint] -> galopiramo ulijevo  dok ne vrijedi */
37       /* a[hint - ofs] < key <= a[hint - lastofs] */
38       const Py_ssize_t maxofs = hint + 1;
39       while (ofs < maxofs) {
40           IFLT(*(a-ofs), key)
41               break; /* key <= a[hint - ofs] */
42
43           lastofs = ofs;
44           ofs = (ofs << 1) + 1;
45           if (ofs <= 0)          /* integer overflow */
46               ofs = maxofs;
47       }
48       if (ofs > maxofs)
49           ofs = maxofs;
51
52      /* Podesi offsetove da budu u odnosu na &a[0]. */
53      /* i pozitivni te zamijeni lastofs i ofs */
54       k = lastofs;
```

```
55       lastofs = hint - ofs;
56       ofs = hint - k;    }
57    a -= hint;
58
59   assert(-1 <= lastofs && lastofs < ofs && ofs <= n);
60
61      /* Sada je a[lastofs] < key <= a[ofs] */
62      /* Binarnim pretraživanjem odredi gdje se */
63      /* točno unutar intervala nalazi key */
64  ++lastofs;
65    while (lastofs < ofs) {
66            /* sredina intervala <lastofs,ofs> */
67        Py_ssize_t m = lastofs + ((ofs - lastofs) >> 1);
68
69        IFLT(a[m], key)
70            lastofs = m+1;        /* a[m] < key */
71        else
72            ofs = m;              /* key <= a[m] */
73    }
74    assert(lastofs == ofs);       /* a[ofs-1] < key <= a[ofs] */
75    return ofs;
76
77      fail:
78             return -1;
79   }
```

`gallop_right` radi kao i `gallop_left`, jedina razlika je što ako u polju ima elemenata koji su jednaki `key`, `gallop_right` će vratiti poziciju *desno* od *najdesnijeg* jednakog elementa (`gallop_left` vraća poziciju *lijevo* od *najljevijeg* jednakog elementa) — zato se zove `gallop_right`. Ta razlika je bitna zbog stabilnosti: pretpostavimo da spajamo tokove *A* i *B*, te je tok *B* viši na stogu. Ako tražimo poziciju elementa `key` iz *A* u *B*, te postoje pojave od `key` u *B*, s obzirom da je u početnom polju instanca `key` iz *A* nastupala prije bilo koje pojave `key` u *B* (jer je *B* viši na stogu), da bi zadržali stabilnost i u spojenom polju, `key` iz *A* mora nastupati prije svih instanci `key` iz *B*, tj. biti lijevo od najljevije pojave `key` u *B*. Zato kada tražimo pozicije elemenata iz *A* u polju *B* koristimo `gallop_left` — u obrnutoj situaciji koristimo `gallop_right`, jer želimo da pojave elementa iz *B* budu desno od instanci tog istog elementa u *A* (vidi stranicu 38). Kôd funkcije se može vidjeti na [11].

merge_lo i merge_hi

Sada ćemo objasniti kako radi funkcija merge_lo. Funkcija kao argumente prima stog i dva sortirana i susjedna toka a i b (b se nalazi iznad a na stogu). ssa i ssb su pozicije tokova a i b, a na i nb su duljine tokova a i b. ms je stog tokova, tj. instanca strukture s_MergeState. Rezultat je spoj tokova a i b — jedan uzlazno sortirani tok na poziciji ssa i duljine na + nb. Funkcija vraća 0 ako je uspjela, a u slučaju greške vraća -1.

Funkcija radi tako da prvo kopira tok a u pomoćnu memoriju i ssa spremi u varijablu dest (*destination*) kao memoriju gdje ćemo spremati sortirane podatke (*MergeMemory*, vidi stranicu 20). Zatim, kopira prvi element toka b u *MergeMemory* (vidi stranicu 43) — te onda kreće s usporedbama. Prvo uspoređujemo elemente „jedan po jedan" i pratimo kada je jedan element bio izabran iz istog toka min_gallop puta za redom (linije 37–76). Znači, usporedimo prvi element iz a — ssa.keys[0] i prvi element iz b — ssb.keys[0] (linija 42). Rezultat usporedbe se sprema u varijablu k — ako je došlo do greške, k == -1 i funkcija skoči na dio fail (linija 45). Ako je k == 1, ssb.keys[0] < ssa.keys[0] i izvršava se blok (if(k)) na 43. liniji. Zapišemo ssb.keys[0] u dest, za 1 povećamo bcount (koliko je puta za redom veći element bio u b), acount stavimo na 0 i smanjimo broj preostalih elemenata u b — nb — za 1 (linije 48–51). Ako je nb == 0, znači da smo kopirali cijelo b polje pa skočimo na succeed (linija 148), a ako je bcount $\geq$ min_gallop, izlazimo iz petlje i krećemo s galopiranjem (linije 82–141). Inače, k == 0 tj. ssa.keys[0] $\leq$ ssb.keys[0] i izvrši se else blok (linije 60–75) koji radi analogno kao u slučaju ssb.keys[0] > ssa.keys[0]. Galopiranje počinje na liniji 82. Galopiramo udesno i tražimo poziciju od ssb.keys[0] u a, krećući od prvog elementa u polju a — broj elemenata u a koji su manji od ssb.keys[0] spremimo u acount i k (ponovno koristimo varijablu k, ovaj put u drugom kontekstu). Ako je k $\neq$ 0 izvršava se blok if (k). Ako je k $<$ 0 tj. k == -1 došlo je do greške pa funkcija skače na fail (linija 150). Inače je $k \geq 1$ pa kopiramo prvih k elemenata iz a u dest i podesimo varijable ssa, dest i na za k elemenata koje smo kopirali (linije 98–101). Nakon toga premjestimo ssb.keys[0] u dest (linija 112), te zatim tražimo poziciju od ssa.keys[0] u b funkcijom gallop_left i analogno kopiramo dio iz b, te nakon toga ssa.keys[0]. Cijeli proces se ponavlja dok je barem jedan od komada iz a ili b koji se kopira veličine barem min_gallop. Ako u nekom trenu u toku a ostane samo jedan element, funkcija skoči na dio CopyB (linija 155, vidi stranicu 43).

```
static Py_ssize_t
merge_lo(MergeState *ms, sortslice ssa, Py_ssize_t na,
         sortslice ssb, Py_ssize_t nb)
{
1    Py_ssize_t k;
2    sortslice dest;/* u dest spremamamo poziciju od a: ssa */
```

```
3    int result = -1;
4    Py_ssize_t min_gallop;
5
6    /* provjera da je ulaz dobro definiran */
7    assert(ms && ssa.keys && ssb.keys && na > 0 && nb > 0);
8    assert(ssa.keys + na == ssb.keys);
9    if (MERGE_GETMEM(ms, na) < 0)/* alociranje pomoćne memorije */
10        return -1;
11
12    /* kopiranje polja a u pomoćnu memoriju */
13    sortslice_memcpy(&ms->a, 0, &ssa, 0, na);
14    dest = ssa;
15    ssa = ms->a;
16
17    /* 1. element u B je manji od 1. elementa u A */
17    sortslice_copy_incr(&dest, &ssb);
19    *dest++ = *pb++;
20    --nb;
21    if (nb == 0)
22        goto Succeed;
23    if (na == 1)
24    /* zadnji element u a pripada na kraj spojenog polja */
25        goto CopyB;
26
27    min_gallop = ms->min_gallop;
28
29    for (;;) {
30
31    /* koliko puta za redom je element iz a/b bio manji */
32        Py_ssize_t acount = 0;
33        Py_ssize_t bcount = 0;
34
35      /* uspoređujemo elemente iz tokova 1 po 1 dok jedan od njih */
36      /* ne sadrži manji element barem min_gallop puta */
37        for (;;) {
38            assert(na > 1 && nb > 0);
39
40      /* usporedba ssb.keys[0] i ssa.keys[0]. k je 1 ako je */
41      /* ssb.keys[0] manji, inače 0. u slučaju greške k je -1 */
```

```
42            k = ISLT(ssb.keys[0], ssa.keys[0]);
43            if (k) {
44                if (k < 0)/* tj. k=-1: ISLT našao grešku */
45                    goto Fail;
46
47      /* element iz b je bio manji-> zapiši i podesi brojače */
48                sortslice_copy_incr(&dest, &ssb);
49                ++bcount;
50                acount = 0;
51                --nb;
52                if (nb == 0) /* kraj toka B, gotovo je spajanje */
53                    goto Succeed;
54
55      /* b je imao manji element u usporedbi barem min_gallop puta */
56      /* pa izlazimo iz while petlje i krećemo galopirati */
57                if (bcount >= min_gallop)
58                   break;
59           }
60           else {
61
62          /* element iz a je bio manji-> zapiši i podesi brojače */
63                sortslice_copy_incr(&dest, &ssa);
64                ++acount;
65                bcount = 0;
66                --na;
67                if (na == 1)
68      /* zadnji element u a pripada na kraj spojenog polja */
69                   goto CopyB;
70
71     /* a je imao manji element u usporedbi barem min_gallop puta */
72     /* pa izlazimo iz petlje i krećemo galopirati */
73                if (acount >= min_gallop)
74                    break;
75           }
76       }
77
78         /* Neki tok je pokrenuo galopiranje */
79
80
```

```
81          ++min_gallop;
82          do {
83              assert(na > 1 && nb > 0);
84
85           /* galopiranje se isplatilo: */
86           /* smanji min_gallop za 1 i ažuriraj ms */
87              min_gallop -= min_gallop > 1;
88              ms->min_gallop = min_gallop;
89
90     /* galopiramo: tražimo poziciju od */
91     /* ssb.keys[0] u a, od početka a */
92              k = gallop_right(ms, ssb.keys[0], ssa.keys, na, 0);
93              acount = k;
94              if (k) {
95                  if (k < 0)
96                      goto Fail;
97           /* kopiraj k elemenata iz a, podesi pointere */
98                  sortslice_memcpy(&dest, 0, &ssa, 0, k);
99                  sortslice_advance(&dest, k);
100                  sortslice_advance(&ssa, k);
101                  na -= k;
102                  if (na == 1)
103           /* zadnji element u a pripada na kraj spojenog polja */
104                    goto CopyB;
105
106
107
108                 if (na == 0)
109                     goto Succeed;
110             }
111          /* kopiraj ssb.keys[0] u sortirani dio, podesi pointere */
112              sortslice_copy_incr(&dest, &ssb);
113              --nb;
114              if (nb == 0)
115                  goto Succeed;
116          /* galopiramo-> tražimo poziciju od */
117          /* ssa.keys[0] u b, od početka b */
118              k = gallop_left(ms, ssa.keys[0], ssb.keys, nb, 0);
119
```

```
120             bcount = k;
121             if (k) {
122                 if (k < 0)
123                     goto Fail;
124          /* kopiraj k elemenata iz b, podesi pointere */
125                 sortslice_memmove(&dest, 0, &ssb, 0, k);
126                 sortslice_advance(&dest, k);
127                 sortslice_advance(&ssb, k);
128                 nb -= k;
129                 if (nb == 0)
130                     goto Succeed;
131             }
132         /* kopiraj ssa.keys[0] u sortirani dio i */
133         /* podesi pointere */
134             sortslice_copy_incr(&dest, &ssa);
135             --na;
136             if (na == 1)
137         /* zadnji element u a pripada na kraj spojenog polja */
138                 goto CopyB;
139
140         /* uvjet galopiranja */
141         } while (acount >= MIN_GALLOP || bcount >= MIN_GALLOP);
142
143         /* prestali smo galopirati: */
144         /* povećaj min_gallop i ažuriraj u strukturi */
145         ++min_gallop;
146         ms->min_gallop = min_gallop;
147     }
148   Succeed:
149      result = 0;
150   Fail:     /* došlo je do greške, kopiraj preostale elemente */
151             /* polja a iz dodatne memorije u */
152    if (na)  /* slobodni dio zajedničke memorije od a i b */
153       sortslice_memcpy(&dest, 0, &ssa, 0, na);
154   return result;
155 CopyB:
156  assert(na == 1 && nb > 0);
157    /* zadnji element u a pripada na kraj spojenog polja */
158    sortslice_memmove(&dest, 0, &ssb, 0, nb);
```

```
159     sortslice_copy(&dest, nb, &ssa, 0);
160     return 0;
161 }
```

Funkcija `merge_hi` radi analogno kao `merge_lo` — jedina razlika je da je u ovom slučaju $|A| > |B|$ pa B kopiramo u dodatnu memoriju i počinjemo spajati tokove od desnog kraja gdje je B bio pohranjen prema lijevo. Kôd funkcije se može vidjeti na [11].

`merge_at`

Funkcija kao argumente prima strukturu za stog `s_MergeState` i indeks `i` koji označava da treba spajati tokove s indeksima `i` (tok A) i `i` + 1 (tok B). Rezultat je jedan tok. Povratna vrijednost je 0 ako je funkcija uspjela, a ako je došlo do greške povratna vrijednost je -1.

Funkcija radi tako da galopiranjem udesno od početka toka A (pozicija `ssa`) utvrdimo gdje se u A nalazi prvi element od B (on se nalazi na poziciji `ssb`). Svi elementi iz A koji se nalaze prije te pozicije možemo ignorirati — s obzirom da spojeni tok spremamo u zajedničku memoriju tokova A i B, oni se već nalaze na svojoj konačnoj poziciji nakon spajanja. Zatim tražimo gdje se u B nalazi zadnji element iz A tako da galopiramo lijevo od posljednje pozicije u B. Elementi iz B koji se nalaze nakon te pozicije su već u konačnoj poziciji pa i njih možemo ignorirati. Rezultat toga su novi tokovi A i B, sa prilagođenim duljinama i početnim pozicijama tako da izbacimo iz njih elemente koji se već nalaze na svojoj konačnoj poziciji. Na kraju te nove tokove spojimo funkcijom `merge_lo`, odnosno `merge_hi`. Bitno je spomenuti da zbog tih prilagodbi, zadnji element novog toka A je veći od svih elemenata novog toka B pa kada ih spajamo znamo da on pripada na posljednju poziciji u spojenom polju. Analogno, prvi element iz novog toka B pripada na prvu poziciju u spojenom polju.

```
static int
merge_at(MergeState *ms, Py_ssize_t i)
{
    sortslice ssa, ssb;
    Py_ssize_t na, nb;
    Py_ssize_t k;

    /* osiguramo da su uvjeti za spajanje dobro postavljeni */
    assert(ms != NULL);
    assert(ms->n >= 2);
    assert(i >= 0);
    assert(i == ms->n - 2 || i == ms->n - 3);
```

```
/* informacije o tokovima se nalaze u strukturi MergeState. */
/* Tokovi se nalaze na polju pending */
/* spremamo duljine tokova u na i nb, a pozicije tokova u ssa i ssb */
ssa = ms->pending[i].base;
na = ms->pending[i].len;
ssb = ms->pending[i+1].base;
nb = ms->pending[i+1].len;
assert(na > 0 && nb > 0);
assert(ssa.keys + na == ssb.keys);

/* zabilježimo duljinu spojenog toka;
 * ako je (i == ms->n - 3) znači da spajamo 2. i 3. tok od vrha,
 * pa treba tok na vrhu (ms->pending[i+2]) pomaknuti za 1 mjesto
 * prema dolje
 */
ms->pending[i].len = na + nb;
if (i == ms->n - 3)
    ms->pending[i+1] = ms->pending[i+2];
--ms->n;

/* Tražimo gdje se 1. element od B nalazi u A, galopirajući
 * od početka. Elementi iz A prije te pozicije su već na
 * svojoj konačnoj poziciji. Podešavamo ssa i na tako da
 * pokazuju na polje bez tih elemenata
 */
k = gallop_right(ms, *ssb.keys, ssa.keys, na, 0);
if (k < 0)
    return -1;
sortslice_advance(&ssa, k);
na -= k;
if (na == 0)
    return 0;

/* Tražimo gdje se zadnji element od A nalazi u B, galopirajući
 * od kraja. Elementi iz B poslije te pozicije su već
 * na svojoj konačnoj poziciji. Podešavamo nb tako da ssb i nb
 * pokazuju na polje B bez tih elementa
 */
```

```
   nb = gallop_left(ms, ssa.keys[na-1], ssb.keys, nb, nb-1);

   /* nb <=0 znači da je ili 0 ili -1. -1 znači da je došlo */
   /* do greške, 0 znači da ssa.keys[na-1] pripada na početak */
   /* polja B, tj. ssa.keys[na-1]<ssb.keys[0], tj. */
   /* polje je već sortirano. */
    if (nb <= 0)
      return nb;

     /* Spajamo polja A i B bez elemenata koji */
     /* su već na svojoj konačnoj poziciji */
    if (na <= nb)
        return merge_lo(ms, ssa, na, ssb, nb);
    else
        return merge_hi(ms, ssa, na, ssb, nb);
}
```

merge_collapse

Sada ćemo objasniti kako radi funkcija `merge_collapse`. Funkcija kao argument prima strukturu za stog `s_MergeState` te provjerava jesu li zadovoljene invarijante na vrhu stoga — ako nisu, spojimo tokove u skladu s definicijom 2.1.1. Povratna vrijednost je 0 u slučaju uspjeha, a -1 u slučaju greške.

```
static int
merge_collapse(MergeState *ms)
{
/* spremamo u p polje tokova pending */
    struct s_slice *p = ms->pending;

    assert(ms);
    while (ms->n > 1) {
/* n služi za provjeru da ima dovoljno tokova na stogu */
        Py_ssize_t n = ms->n - 2;

/* provjera je li prekršena invarijanta 1 ili 2 */
        if ((n > 0 && p[n-1].len <= p[n].len + p[n+1].len)||
```

```
            ( n > 1 && p[n-2].len <= p[n-1].len + p[n].len))) {

        /* 2. tok od vrha se spaja s manjim od 1. i 3. */
            if (p[n-1].len < p[n+1].len)
                --n;
            if (merge_at(ms, n) < 0)
                return -1;
        }
        /* prekršena je invarijanta 3 */
        else if (p[n].len <= p[n+1].len) {
                 if (merge_at(ms, n) < 0)
                        return -1;
        }
        else /* obje invarijante su zadovoljene */
            break;
    }
    return 0;
}
```

merge_force_collapse

Funkcija `merge_force_collapse` kao argument prima strukturu za stog `MergeState` u varijabli `ms`. Nju koristimo nakon što su svi tokovi stigli na stog i zadovoljene su invarijante. Spojimo sve tokove na stogu dok ne ostane samo jedan tok — to je traženo sortirano polje. Tokeve spajamo tako da tok na drugoj poziciji od vrha stoga spojimo s manjim od prvog i trećeg toka na vrhu stoga. Povratna vrijednost je 0 u slučaju uspjeha, a -1 u slučaju greške.

```
static int
merge_force_collapse(MergeState *ms)
{
    struct s_slice *p = ms->pending;

    assert(ms);
    while (ms->n > 1) {

        /* n služi za provjeru da ima dovoljno tokova na stogu */
```

```
        Py_ssize_t n = ms->n - 2;
        if (n > 0 && p[n-1].len < p[n+1].len)/* 2. tok od vrha se */
            --n;                        /* spaja s manjim od 1. i 3. */
        if (merge_at(ms, n) < 0)
            return -1;
    }
    return 0;
}
```

merge_compute_minrun

Funkcija `merge_compute_minrun` kao argument prima duljinu početnog polja `n`. Povratna vrijednost je `minrun`.

Funkcija radi tako da u `while` petlji `n` pomiče udesno za 1 bit, tj. `n` cjelobrojno dijeli s 2 sve dok je $n \geq 64$. Pretpostavimo da smo imali k pomaka. Funkcija vraća završnu vrijednost od n ($\left\lfloor \frac{n}{2^k} \right\rfloor$) uvećanu za 1 ako je pri pomicanju bilo ostatka ($2^k \nmid n$). Sada ćemo pokazati da vrijedi $\left\lfloor \frac{N}{\texttt{minrun}} \right\rfloor \leq 2^k$. Pretpostavimo da smo `n` pomaknuli udesno k puta. Ako nismo shiftali ni jednu jedinicu, tada je $\texttt{minrun} = \left\lfloor \frac{N}{2^k} \right\rfloor$ (N smo djelili k puta s 2, bez ostatka) pa slijedi da je $\left\lfloor \frac{N}{\texttt{minrun}} \right\rfloor = 2^k$. Ako smo pomaknuli udesno barem jednu jedinicu. vrijedi $\texttt{minrun} = \texttt{n'} + 1$. Da bi pokazali $\left\lfloor \frac{N}{\texttt{minrun}} \right\rfloor \leq 2^k$, ekvivalentno je da pokažemo $\left\lfloor \frac{N}{2^k} \right\rfloor \leq \texttt{minrun}$, tj. $\left\lfloor \frac{N}{2^k} \right\rfloor \leq \texttt{n'} + 1$.

$$
\begin{aligned}
\left\lfloor \frac{N}{2^k} \right\rfloor &\leq \frac{N}{2^k} \\
\frac{N}{2^k} &\leq n' + 1 \\
N &\leq n' \cdot 2^k + 2^k \\
N - n' \cdot 2^k &\leq 2^k \\
[\text{jer } N - n' \cdot 2^k &\leq 2^{k-1}(1 + 1/2 + 1/4 \ldots + 1/2^{k-1})] \qquad (2.9) \\
2^{k-1}(1 + 1/2 + 1/4 \ldots + 1/2^{k-1}) &\leq 2^k \\
(1 + 1/2 + 1/4 \ldots + 1/2^{k-1}) &\leq 2
\end{aligned}
$$

Nejednakost (2.9) vrijedi jer smo n' dobili tako da smo N cjelobrojno dijelili s 2 k puta. Kada n' pomnožimo s 2^k, rezultat je N takav da su mu bitovi značajnosti manje od k

jednaki 0. Znači, razlika $N - n' \cdot 2^k$ će biti N takav da su mu svi bitove značajnosti k i više postavljeni na 0, tj. vrijedi $N - n' \cdot 2^k \leq 2^{k-1}(1 + 1/2 + 1/4 \ldots + 1/2^{k-1})$.

Znači, vrijedi $\left\lfloor \frac{N}{\texttt{minrun}} \right\rfloor \leq 2^k$ i vidimo da kad `minrun` nije potencija od dva, $\left\lfloor \frac{N}{\texttt{minrun}} \right\rfloor$ je strogo manje od potencije od dva ali blizu potencije od dva.

```
static Py_ssize_t
merge_compute_minrun(Py_ssize_t n)
{
    Py_ssize_t r = 0;/* ako pomaknemo jedinicu, r je 1, inače je 0 */

    assert(n >= 0);
    while (n >= 64) {
        r |= n & 1; /* ako je n neparan, r postaje 1 */
        n >>= 1;    /* n pomičemo za 1 bit */
    }
    return n + r;
}
```

`list_sort_impl`

U ovom odjeljku objašnjavamo kako radi funkcija `list_sort_impl`. To je funkcija koju Python poziva kada želimo sortirati polje. Kao argumente prima `self` — polje koje želimo sortirati.

Funkcija radi tako da početno polje `self` podijeli u tokove — krećemo od početka polja i identificiramo prvi tok. Kada smo ga našli, stavimo ga na stog i pozovemo funkciju `merge_collapse` koja provjerava invarijante i u skladu s njima spaja tokove na stogu. Nakon toga identificiramo sljedeći tok i proces se ponavlja dok ne dođemo do kraja polja — tada, ako na stogu postoji više od jednog toka, pozove se funkcija `merge_force_collapse` koja od vrha prema dnu stoga spaja sve tokove, dok nam na stogu ne preostane samo jedan tok — to je traženo sortirano polje. U slučaju uspjeha funkcija vrati `Py_None`, a u slučaju greške `NULL`.

Prikazat ćemo samo dio koda koji je bitan za razumijevanje rada algoritma — cijela funkcija se može vidjeti na [11]. Dijelove koda koji nedostaju ćemo označiti /* ... */.

```
static PyObject *
list_sort_impl(PyListObject *self, PyObject *keyfunc, int reverse)
{
    MergeState ms;
    Py_ssize_t nremaining;
```

```
Py_ssize_t minrun;
sortslice lo;
Py_ssize_t saved_ob_size, saved_allocated;
PyObject **saved_ob_item;
PyObject **final_ob_item;

/* ako je došlo do greške, varijabla result će ostat NULL */
PyObject *result = NULL;
Py_ssize_t i;
PyObject **keys;

/* ... */

/* veličina ulaznog polja */
saved_ob_size = Py_SIZE(self);

/* adresa ulaznog polja */
saved_ob_item = self->ob_item;

/* ... */

/* ulazno polje se sprema u varijablu lo */
if (keyfunc == NULL) {
    keys = NULL;
    lo.keys = saved_ob_item;
    lo.values = NULL;
}
else {

/* ... */

    lo.keys = keys;
    lo.values = saved_ob_item;
}


/* ... */

/* poziva se kontruktor za stog ms */
```

```
 merge_init(&ms, saved_ob_size, keys != NULL);

/* broji elemente koji nisu dio nekog toka */
nremaining = saved_ob_size;

/* na početku je to veličina cijelog polja(saved_ob_size) */
if (nremaining < 2)
    goto succeed;

/* ... */

minrun = merge_compute_minrun(nremaining);

do {
    int descending;
    Py_ssize_t n;

    /* identificiramo sljedeći tok */
    n = count_run(&ms, lo.keys, lo.keys + nremaining, &descending);
    if (n < 0)
        goto fail;

    /* ako je strogo padajući, obrnemo ga */
    if (descending)
        reverse_sortslice(&lo, n);

    /* ako je manji od minrun, nadopuni do minrun s binarysort */
    if (n < minrun) {
        const Py_ssize_t force = nremaining <= minrun ?
                          nremaining : minrun;
        if (binarysort(&ms, lo, lo.keys + force, lo.keys + n) < 0)
            goto fail;
        n = force;
    }

    assert(ms.n < MAX_MERGE_PENDING);

    /* spremamo tok na stog */
    ms.pending[ms.n].base = lo;
```

```
        ms.pending[ms.n].len = n;
        ++ms.n;

        /* provjeri invarijante i spoji ih po definiciji */
        if (merge_collapse(&ms) < 0)
            goto fail;

        /* pomakni pointer na sljedeći element koji nije u toku */
        sortslice_advance(&lo, n);
        nremaining -= n;

    /* ponavljaj dok postoji barem 1 element */
    /* koji nije u nekom toku */
    } while (nremaining);


    /* ako ima više od jednog toka spoji ih sve */
    if (merge_force_collapse(&ms) < 0)
        goto fail;

     /* ... */

succeed:
    result = Py_None;
fail:

    /* ... */

    return result;
}
```

2.3 Složenost

U ovom odjeljku ćemo dokazati da *Timsort* ima vremensku složenost $O(n \log n)$ u najgorem slučaju. Dokaz je prilagođen s [5].

Teorem 2.3.1. *Vremenska složenost Timsorta je* $O(n \log n)$.

Dokaz. Prvi korak dokaza je *Timsort* prikazati u pseudokodu. To će nam olakšati analizu rada algoritma.

```
Data: A sequence S to sort
Result: The sequence S is sorted into a single run, which remains on the stack.
runs ⟵ run decomposition of S;
T ⟵ an empty stack;
h⟵ height(T);
while runs ≠ ∅ do // glavna petlja Timsorta
    remove a run t from runs and push t onto T;
    merge_collapse
end while
if h ≠ 1 then
    merge_force_collapse(T)
end if
```

Algoritam 1: *Timsort*

```
Data: A stack of runs T
Result: Fibonaccijevo svojstvo i invarijanta (3) su zadovoljeni
while height(T)> 1 do
    n ⟵ height(T)−2
    if (n > 0 and t3 ≤ t2 + t1 ) or (n > 1 and t4 ≤ t3 + t2) then
        if t3 < t1 then
            merge runs T2 and T3 on the stack
        else
            merge runs T1 and T2 on the stack
        end if
    else if t2 ≤ t1 then
        merge runs T1 and T2 on the stack
    else
        break
    end if
end while
```

Algoritam 2: funkcija `merge_collapse`

Algoritmi 1 i 2 zajedno predstavljaju *Timsort*. Algoritam 1 je pseudokod za `list_sort_impl`, tj. funkciju koja obavlja sortiranje, a Algoritam 2 je pseudokod za `merge_collapse` — pomoćnu funkciju koju `list_sort_impl` poziva. Radi lakše analize rada, zapisat ćemo algoritme 1 i 2 kao jedan algoritam — algoritam 3, čiji pseudokod možemo vidjeti na sljedećoj slici. Invarijanta (2), za koju se u radu [6] shvatilo da je potrebna za ispravno funkcioniranje algoritma, je podcrtana.

Data: A sequence S to sort
Result: The sequence S is sorted into a single run, which remains on the stack.

```
1  runs ⟵ run decomposition of S;
2  T ⟵ an empty stack;
3  h⟵ height(T);
4  while runs ≠ ∅ do // glavna petlja Timsorta
5      remove a run t from runs and push t onto T;                    // #1
6      while true do
7          if h ≥ 3 and t1 > t3 then
8              merge the runs T2 and T3;                              // #2
9          else if h ≥ 2 and t1 ≥ t2 then
10             merge the runs T1 and T2;                              // #3
11         else if h ≥ 3 and t1 + t2 ≥ t3 then
12             merge the runs T1 and T2;                              // #4
13         else if h ≥ 4 and t2 + t3 ≥ t4 then
14             merge the runs T1 and T2;                              // #5
15         else
16             break
17         end if
18     end while
19 end while
20 while h ≠ 1 do
21     merge the runs T1 and T2
22 end while
```

Algoritam 3: algoritmi 1 i 2 zapisani kao jedan algoritam

Propozicija 2.3.2. *Za svaki ulaz, algoritmi 1 i 3 rade iste usporedbe.*

Dokaz. Jedina razlika između algoritama je da smo algoritam 2 pretvorili u `while` petlju algoritma 3 (linije 6–18). Da algoritmi rade iste usporedbe možemo provjeriti tako da promatramo različite slučajeve za spajanje. Ako je $t_3 < t_1$ onda je i $t_3 \leq t_1 + t_2$, tj. izvrši se linija 5 u algoritmu 2 i linija 8 u algoritmu 3 pa oba algoritma spajaju drugi i treći tok od vrha. Inače, ako je $t_3 \geq t_1$ tada oba algoritma spajaju prvi i drugi tok ako i samo ako je zadovoljen jedan od tri uvjeta: $t_2 \leq t_1$, $t_3 \leq t_1 + t_2$ ili $t_4 \leq t_2 + t_3$. □

Da bismo dokazali Teorem (2.3.1) potrebno je analizirati samo glavnu petlju algoritma 3 (linije 4–19). Rastavljanje ulaznih podataka u tokove se može napraviti u linearnom vremenu, a zadnju petlju (linija 20) bi mogla odraditi glavna petlja tako da dodamo fiktivni tok duljine $n + 1$ na kraj stoga koji sadrži tokove. Također, radi jednostavnosti dokaza nećemo provjeravati je li h dovoljno velik u slučajevima #2 do #5 — kada bi redom dodali tokove duljina $8n$, $4n$, $2n$, i n na stog prije rastavljanja ulaznih podataka u tokove osigurali bi da je $h \geq 4$ za vrijeme izvršavanja glavne petlje. To sve bi povećalo n (broj elemenata) linearno (najviše 32 puta), pa ne bi utjecalo na konačnu ocjenu složenosti u ovisnosti o n. Sada ćemo iznijeti glavnu ideju dokaza. Svakom elementu ulaznog polja dodjeljujemo tokene koje koristimo za uspoređivanje elemenata. Za svaku usporedbu moramo platiti s dva tokena, te svakom elementu dodjeljujemo $O(\log n)$ tokena. Pokazat ćemo da je broj tokena koji je elementu preostao uvijek nenegativan, iz čega ćemo moći zaključiti da smo napravili najviše $O(n \log n)$ usporedbi.

Definiramo *visinu* elementa ulaznog polja kao broj tokova koji se nalaze ispod njega: elementi koji pripadaju toku T_i na stogu $(T_1, \ldots, T_h)$ imaju visinu $h - i$ (T_h nam označava trenutni tok na *dnu* stoga, a T_1 trenutni tok na *vrhu* stoga). Također, sa t_i označavamo duljinu toka T_i. Radi jednostavnosti, definiramo dvije vrste tokena: $\heartsuit$ i $\diamond$. Obje vrste tokena se mogu koristiti za plaćanje usporedbi.

Dva tokena $\diamond$ i jedan token $\heartsuit$ su dodijeljeni elementu kada njegov tok dođe na stog (slučaj #1 u algoritmu 3) ili kada mu se visina smanji — svim elementima toka T_1 su dodjeljeni tokeni kada se spoje T_1 i T_2, tj. svim elementima tokova T_1 i T_2 su dodjeljeni tokeni kada se spoje T_2 i T_3. Kao što smo spomenuli, tokene koristimo da platimo za usporedbe, ovisno o tome koji od slučajeva #2 do #5 u algoritmu 3 je nastupio:

- Slučaj #2: svaki element od T_1 i T_2 plaća $1\,\diamond$. To je dovoljno da pokrijemo cijenu spajanja T_2 i T_3 jer je u ovom slučaju $t_3 < t_1$, pa vrijedi $t_2 + t_3 \leq t_1 + t_2$.
- Slučaj #3: svaki element od T_1 plaća $2\,\diamond$. O ovom slučaju je $t_1 \geq t_2$, pa je $t_1 + t_2 \leq 2t_1$.
- Slučajevi #4 i #5: svaki element od T_1 plaća $1\,\diamond$, a svaki element od T_2 plaća $1\,\heartsuit$. Cijena spajanja je $t_1 + t_2$, a to je upravo broj tokena koje smo potrošili.

Propozicija 2.3.3. *Za vrijeme izvršavanje glavne petlje Timsorta, količina tokena $\diamond$ i $\heartsuit$ za svaki element je nenegativna.*

Dokaz. S obzirom da se u slučajevima #2 do #5 visina elemenata toka T_1 smanjila, a u slučaju #2 i visina elemenata toka T_2, u oba toka broj dodjeljenih tokena $\diamond$ je veći ili jednak broju potrošenih tokena $\diamond$, tj. za svaki element količina tokena $\diamond$ je nenegativna. Tokenima $\heartsuit$ se plaća samo u slučajevima #4 i #5 — svaki element toka T_2 potroši jedan token $\heartsuit$ i pripada toku $\overline{T}_1$ novog stoga $\overline{S} = (\overline{T}_1, \ldots, \overline{T}_{h-1})$. S obzirom da je $\overline{T}_i = T_{i+1}$ za $i \geq 2$, preduvjet za slučaj #4 implicira da je $t_1 \geq t_2$, a preduvjet za slučaj #5 da je $t_1 + t_2 \geq t_3$.

U oba slučaja sljedeći korak u algoritmu je još jedno spajanje na stogu $\overline{S}$, čime se smanji visina toka $\overline{T}_1$, među kojima se nalaze i elementi koji su pripadali toku T_2 pa po našoj strategiji dodjeljivanja tokena tim elementima se dodjeli jedan token ♡ (i dva tokena ♢) a ne izgube ni jedan token ♡ jer prema našoj strategiji najviši tok na stogu nikada ne plaća s tokenima ♡. Time smo dokazali da kadgod neki element plati tokenom ♡, sljedeći korak algoritma je još jedno spajanje u kojem taj element nadoknadi potrošeni token ♡. □

Označimo s h_{max} maksimalni broj tokova na stogu za vrijeme izvršavanja algoritma. Zbog načina na koji dodjeljujemo tokene, svakom elementu polja je dodjeljeno najviše $2h_{max}$ tokena ♢ i h_{max} tokena ♡. Znači da bi dokazali teorem treba još pokazati da je $h_{max} = O(\log n)$. Da bi to dokazali, u sljedećoj propoziciji dokazujemo četiri invarijante koje vrijede u svakom koraku glavne petlje *Timsorta* (tj. algoritma 3).

U sljedećoj propoziciji uvjet (*i*), osim za indekse $i \in \{1, 2\}$, odgovara Fibbonacijevom svojstvu (vidi stranicu 19). Uvjeti (*ii*) i (*iii*) su slični invarijanti (3) (vidi stranicu 17). Uvjet (*ii*) kaže da je duljina drugog tok od vrha manja od tri puta duljine trećeg toka, a uvjet (*iii*) da je duljina trećeg toka od vrha manja od duljine četvrtog toka od vrha. Invarijanta (3) kaže da je duljina toka na vrhu manja od duljine drugog toka od vrha.

Propozicija 2.3.4. *U svakom koraku glavne petlje Timsorta vrijedi:* $(i)\, t_i + t_{i+1} < t_{i+2}$ *za* $i \in \{3, \ldots, h-2\}$, $(ii)\, t_2 < 3t_3$, $(iii)\, t_3 < t_4$, $(iv)\, t_2 < t_3 + t_4$.

Dokaz. Dokaz ćemo odraditi indukcijom — dokazat ćemo da ako invarijante vrijede u nekom koraku algoritma, da tada vrijede i kada se stog S promijeni kao rezultat jednog od slučaja #1 do #5. Označimo sa $\overline{S} = (\overline{T}_1, \ldots, \overline{T}_{\overline{h}})$ novo stanje stoga i provjeravamo invarijante ovisno o slučaju, od #1 do #5, koji je doveo do novog stanja stoga:

- Ako je nastupio slučaj #1, tada je novi tok došao na stog — $\overline{T}_1$. To znači da na stogu S nije vrijedio nijedan od uvjeta #2 do #5 (jer bi u tom slučaju se nastavili spajati tokovi na stogu), tj. vrijedi $t_1 < t_2 < t_3$ i $t_2 + t_3 < t_4$ (t_1 i t_2 su strogo manji od t_3 jer bi u suprotnom, s obzirom da je svaki tok duljine barem dva, vrijedilo $t_1 + t_2 > t_3$). S obzirom da je $\bar{t}_i = t_{i-1}$ za $i \geq 2$ i invarijanta (*i*) je zadovoljena za S, slijedi da je $\bar{t}_2 < \bar{t}_3 < \bar{t}_4$ — otuda se lako vidi da su invarijante $(i) - (iv)$ zadovoljene za $\overline{S}$.

- Ako je nastupio jedan od slučajeva #2 do #5, vrijedi da je $\bar{t}_2 = t_2 + t_3$ (slučaj #2) ili $\bar{t}_2 = t_3$ (slučajevi #3 do #5). Koji god slučaj da je nastupio, možemo zaključiti da je $\bar{t}_2 \leq t_2 + t_3$. S obzirom da je $\bar{t}_i = t_{i+1}$ za $i \geq 3$ i invarijante $(i) - (iv)$ su zadovoljene za S, slijedi

  · $\bar{t}_2 \leq t_2 + t_3 < (t_3 + t_4) + t_3 < 3t_4 = 3\bar{t}_3$

  · $\bar{t}_3 = t_4 < t_3 + t_4 < t_5 = \bar{t}_4$

  · $\bar{t}_2 \leq t_2 + t_3 < (t_3 + t_4) + t_3 < t_3 + t_5 < t_4 + t_5 = \bar{t}_3 + \bar{t}_4$

tj. zadovoljeni su uvjeti (*i*)–(*iv*) za $\overline{S}$. □

Koristeći ovu propoziciju, možemo dokazati da je za vrijeme izvršavanja algoritma visina stoga ograničena sa $O(\log n)$.

Propozicija 2.3.5. *U bilo kojem trenutku izvršavanja glavne petlje Timsorta, za stog* $(T_1, \ldots, T_h)$ *vrijedi* $t_2/3 < t_3 < t_4 < \ldots < t_h$ *te za sve* $i \geq j \geq 3$*, vrijedi* $t_i > \sqrt{2}^{i-j-1} t_j$*, iz čega slijedi da je broj tokova na stogu uvijek reda veličine* $O(\log n)$.

Dokaz. Prema prethodnoj propoziciji vrijedi $t_i + t_{i+1} < t_{i+2}$ za $3 \leq i \leq h-2$, tj. vrijedi $t_{i+2} - t_{i+1} > t_i > 0$, iz čega slijedi da vrijedi $t_4 < t_5 < t_6 < \ldots < t_h$. U kombinaciji s invarijantama (*ii*) i (*iii*) iz prethodne propozicije dobijemo prvu tvrdnju ove propozicije, iz koje zatim slijedi da za $j \geq 3$ vrijedi $t_{j+2} > 2t_j$, iz čega slijedi da je $t_i > \sqrt{2}^{i-j-1} t_j$. tj., ako je $h \geq 3$, tada $t_h > \sqrt{2}^{h-4} t_3$, iz čega slijedi, s obzirom da je $t_h \leq n$, da je visina stoga h reda veličine $O(\log n)$. □

Da bismo dovršili dokaz teorema, dovoljno je osvrnuti se na sve tvrdnje koje smo dokazali. Kao što smo već spomenuli, rastavljanje ulaznih podataka u tokove se može napraviti u linearnom vremenu. Zatim smo dokazali da je glavnoj petlji potrebno $O(nh_{max})$ usporedbi tako da samo pokazali da broj tokena koji se dodjeljuju elementima ograničen odozgo, te smo dokazali da duljine tokova rastu eksponencijalnom brzinom, što za posljedicu ima da je $h_{max} = O(\log n)$. Na kraju, spajanje koje se obavlja na liniji 11 možemo riješiti dodavanjem fiktivnog toka od n+1 elemenata, kako smo već objasnili, a možemo i direktno: pretpostavimo da nam je trenutna situacija na stogu $S = (T_1, \ldots, T_h)$. U završnoj petlji svaki element toka T_i sudjeluje u najviše $h + 1 - i$ usporedbi, iz čega slijedi da je ukupna cijena spajanja u liniji 11 $O(n \log n)$. Znači, ukupan broj usporedbi potreban za izvršavanje čitavog algoritma je $O(n \log n)$. □

Bibliografija

[1] 2018. URL: https://en.wikipedia.org/wiki/Herman_Hollerith.

[2] 2018. URL: https://en.wikipedia.org/wiki/Sorting.

[3] 2018. URL: http://www.computerscijournal.org/vol7no3/root-to-fruit-2-an-evolutionary-approach-for-sorting-algorithms/.

[4] 2018. URL: https://en.wikipedia.org/wiki/Timsort.

[5] Nicolas Auger i dr. „On the Worst-Case Complexity of TimSort". *arXiv preprint arXiv:1805.08612* (2018).

[6] Stijn de Gouw. 2015. URL: http://envisage-project.eu/proving-android-java-and-python-sorting-algorithm-is-broken-and-how-to-fix-it/.

[7] George T. Heineman, Gary Pollice i Stanley Selkow. *Algorithms in a nutshell: a practical guide*. O'Reilly Media, Inc., 2016.

[8] Donald Ervin Knuth. *The art of computer programming: sorting and searching*. Sv. 3. Pearson Education, 1997.

[9] Donald Ervin Knuth. *The art of computer programming, vol 1: Fundamental algorithms*. 1973.

[10] Tim Peters. 2002. URL: https://svn.python.org/projects/python/trunk/Objects/listsort.txt.

[11] Tim Peters. 2002. URL: https://github.com/python/cpython/blob/master/Objects/listobject.c.

Sažetak

Timsort je jedan od najkorištenijih algoritama danas te ga razni programski jezici koriste kao standardni algoritam za sortiranje. U uvodnom dijelu dajemo kratki opis problema sortiranja i uvodimo određene pojmove vezane za sortiranje koji nam koriste da bismo lakše raspravljali o svojstvima *Timsorta*. Zatim dokazujemo donju granicu za složenost uspoređujućih algoritama i prikazujemo povijest algoritama za sortiranje, te neke od njih detaljnije opisujemo. Glavni dio rada se sastoji od ilustriranja načina rada i dokaza složenosti *Timsorta*. Prvo opisujemo kako radi algoritam, zatim na izvornom kodu detaljno objašnjavamo sve funkcije i svaki korak algoritma. Na kraju dajemo dokaz $O(n \log n)$ složenosti *Timsorta*, što je najbolja moguća složenost za uspoređujuće algoritme.

Summary

Timsort is one of the most widely used algorithms today. For many programming languages it is the standard sorting algorithm used. In the introductory part a short description of the problem of sorting is presented and we introduce certain definitions related to sorting that help us discuss some of *Timsort's* properties. We then provide a proof of the lower bound for the complexity of comparison sort algorithms and present the history of sorting algorithms, some of which we explain in further detail. The main part of the thesis is comprised of illustrating the sorting process of *Timsort* and a proof of its complexity. First, we describe how the algorithm works. Then, using the source code, we explain in further detail all the functions and every step of the algorithm. Lastly, we provide a proof for *Timsort's* $O(n \log n)$ complexity, which is the best possible for comparison sort algorithms.

Životopis

Osobni podaci:

- Ime i prezime: Mislav Beg
- Datum rođenja: 7. veljače 1994.
- Mjesto rođenja: Zagreb, Republika Hrvatska

Obrazovanje:

- 2016.–2019. Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet, Matematički odsjek, diplomski studij Računarstvo i matematika
- 2013.–2016. Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet, Matematički odsjek, preddiplomski studij Matematika, smjer inženjerski
- 2009.–2013. V. gimnazija, Zagreb
- 2001.–2009. Osnovna škola Petar Zrinski, Zagreb

Zvanje:

- rujan 2016. „Sveučilišni prvostupnik matematike”, Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet, Matematički odsjek, preddiplomski studij Matematika, smjer inženjerski