Analiza algoritmilor elementari de sortare

Paul Popa
Universitatea de Vest Timișoara
Facultatea de Matematică și Informatică
Departamentul de Informatică
Email: paul.popa99@e-uvt.ro

aprilie, 2019

Rezumat

Algoritmii de sortare reprezintă o adevărată provocare în lumea informaticii, ei având o arie de aplicabilitate suficient de mare. Interesul este sortarea cât mai eficientă a datelor, iar pentru acest lucru este utilă cunoaşterea algoritmilor elementari de sortare (Bubble Sort, Insertion Sort, Selection Sort, QuickSort, Merge Sort). De la aceşti algoritmi de bază pornesc ideile pentru algoritmii complexi de sortare. În acest articol vom prezenta idea din spatele ficărui algoritm elementar de sortare, analiza complexităţii în timp şi vom implementa algoritmii pentru a-i putea testa pe seturi de date mici, medii si mari. La final vom vedea care dintre algoritmi este mai potrivit decât ceilalţi în funcţie de situaţie şi dacă există sau nu cazuri deosebite în care algoritmii se comportă diferit de analiza teoretică.

Cuprins

1 Introducere

Un algoritm nu este altceva decât o serie de pași elementari (număr finit) care au ca scop rezolvarea unei probleme sau categorii de probleme. Acesta poate fi aplicat de oricine, rezultatul fiind același dacă algoritmul este aplicat pe aceeași problemă și în aceleași condiții.

Printre algoritmii importanți ai informaticii se numără algoritmii elementari de sortare (Bubble Sort, Insertion Sort, Selection Sort, Quick Sort, Merge Sort) care au diferite aplicații în viața de zi cu zi. Spre exemplu, este mai ușor să căutăm în agenda de contacte a telefonului dacă aceasta este sortată decât dacă contactele au o ordine complet aleatorie. Un alt exemplu ar fi ordonarea studenților după media lor pe semestrul precedent pentru a fi mai ușor de văzut care sunt primii n studenți care primesc bursă de merit pe semestrul curent. Din nou, ar fi destul de dificil să căutăm primii n studenți în ordinea burselor dacă lista cu studenții nu ar fi ordonată după acest criteriu. Desigur că o problemă de dimensiuni mici poate fi direct rezolvată fără ajutorul calculatorului sau a unui algoritm de sortare, însă pentru un număr mare de date (ex: de ordinul sutelor, miilor etc.) acești algoritmi sunt recomandați.

De asemenea, un număr de date de ordinul sutelor poate fi ordonat folosind oricare dintre algoritmii elementari de sortare, deoarece diferența dintre timpii de sortare necesari este prea mică. Problema utilizării unui algoritm anume se pune în momentul sortării unor date de intrare de ordinul miilor, milioanelor etc. În acel moment trebuie să alegem algoritmul potrivit în funcție de tipul datelor ce urmează a fi sortate. Spre exemplu, într-un caz poate fi util sa folosim Insertion Sort, dar poate în alt caz poate fi util sa folosim Selection Sort.

2 Analiza teoretică a algoritmilor elementari de sortare

În această parte a articolului vom analiza algoritmii elementari de sortare din punct de vedere al timpului de execuţie. Acest lucru presupune determinarea a două informaţii importante, dimensiunea problemei şi operaţia dominantă. Intuitiv, dimensiunea problemei este reprezentată de dimensiunea datelor de intrare. De exemplu, dacă avem un tablou unidimensional de n elemente, acest lucru va determina dimensiunea problemei ca fiind egală cu n. În cazul nostru, toţi algoritmii vor avea dimensiunea problemei n, deci nu vom scrie această informaţie de fiecare dată. Aproape la fel de intuitiv, operaţia dominantă se referă la operaţia care se execută de cele mai multe

ori pe parcursul unui algoritm. În cazul în care se afla mai multe astfel de operaţii, vom considera doar una dintre ele. Operaţiile care pot fi alese ca operaţie dominantă sunt împarţite în două categorii. Prima categorie o reprezintă operaţiile de comparaţie ($=$, $\neq$, $<$, $\leq$, $>$, $\geq$) care sunt considerate echivalente şi sunt des întâlnite în cadrul algoritmilor de căutare şi sortare. În a doua categorie intră operaţiile aritmetice care, la rândul lor, se împart în două subcategorii. Prima subcategorie este cea a celor aditive, mai exact adunarea, scăderea, incrementarea si decrementarea. A doua subcategorie este formată din operaţiile multiplicative, mai exact înmulţirea, împărţirea şi restul împărţirii a două numere (mod sau %). Operaţiile aditive sunt considerate, de regulă, mai rapide decât operaţiile multiplicative. Desigur, există şi o a treia categorie de operaţii formată din funcţii geometrice şi logaritmice, operaţii care consumă mai mult timp decât operaţiile multiplicative, însă această categorie este în afara scopului acestei lucrări. Indiferent de categorie, noi vom considera că aceste operaţii se execută într-o unitate de timp pentru a simplifica lucrurile şi pentru a ne axa strict pe ordinul de creştere. Totodată, vom extrage din calcule doar termenul de rang maxim, el fiind termenul dominant din ecuaţie.

Pentru a putea determina eficienţa unui algoritm este necesar să considerăm două cazuri posibile ale datelor de intrare. Aceste două cazuri sunt cazul favorabil si cazul nefavorabil. Intuitiv, pentru cazul favorabil va trebui sa condiderăm setul de date de intrare care va face algoritmul să se execute în cel mai scurt timp, iar pentru cazul nefavorabil să considerăm setul de date de intrare care va face algoritmul să se execute în cel mai lung timp. În continuare vom nota timpul de execuţie cu $T(n)$. Pentru a putea grupa algoritmii în clase în funcţie de ordinul de creştere al timpului de execuţie vom folosi două notaţii, numite notaţii asimptotice. După convenţie, dacă ne referim la cazul favorabil vom folosi notaţia Ω, iar dacă ne referim la cazul nefavorabil vom folosi notaţia O. Dacă nu există caz favorabil şi nefavorabil, folosim notaţia Θ. În unele cazuri este posibil să fie util calculul timpului mediu de execuţie. Pentru acest calcul vom folosi notaţia $T_{mediu}(n)$ şi vom calcula timpul mediu de execuţie după formula (1). Totodată, este necesar să găsim cele k clase de date ce pot servi drept date de intrare. Astfel, $P_i(n)$ reprezintă probabilitatea existenţei clasei i de date ca date de intrare. De regulă această probabilitate este constantă, fiind egală cu $\frac{1}{k}$. După cum am precizat mai sus, $T_i(n)$ este timpul de execuţie pentru clasa i de date de intrare.

$$T_{mediu}(n) = \sum_{i=1}^{k} P_i(n) T_i(n) \tag{1}$$

Bubble Sort

Idee: Bubble Sort, vezi *Algorithm 1*, este cel mai simplu algoritm de sortare care se bazeaza pe interchimbarea elementelor vecine care nu sunt în ordinea dorită (crescătoare sau descrescătoare) până când tot setul de date este sortat.

Simplitatea algoritmului vine din faptul că acesta se inspiră din realitate. Dacă am sacrifica două minute pentru a ne întreba bunica cum sortează cartofii din găleată, răspunsul va fi foarte simplu: bunica ia găleata şi o scutură, astfel cei mai mici cartofi se vor strecura prin spaţiul restrâns şi vor ajunge la fund, iar cei mai mari cartofi vor rămâne în vârf. Dacă acest exemplu nu ne convinge, atunci putem să considerăm drept exemplu un pahar cu o băutură acidulată. Dacă observăm cu atenţie vom vedea cum bulele mari stau deasupra, iar bulele mici stau mai jos.

Despre inventatorul algoritmului nu se cunosc detalii. D. Knuth oferă o referinţă a analizei algoritmului în cartea sa, dar atât. Prima apariţie în scris a "Bubble Sort" a fost făcută de Iverson în 1962. Până atunci au fost făcute multe referiri la acelaşi algoritm, dar cu numele de "sorting by exchange" sau "exchange sorting".

Acestea fiind spuse, este timpul să trecem la analiza din punct de vedere al timpului de execuţie. Dacă considerăm drept operaţie dominantă comparaţia $a[j] > a[j+1]$ vom obţine:

$$T(n) = \sum_{i=1}^{n} \sum_{j=1}^{n-1} 1 = \sum_{i=1}^{n} (n-1) = n(n-1) = n^2 - n \approx n^2$$

Putem observa că pentru această implementare a Bubble Sort nu există caz favorabil si nefavorabil. Deci, $T(n) \in \Theta(n^2)$. Altfel spus, algoritmul are o creştere pătratică, indiferent de datele de intrare.

Algorithm 1 Bubble Sort

```
1: function SORT(a[1..n])
2:     for i ← 1, n do
3:         for j ← 1, n − 1 do
4:             if a[j] > a[j + 1] then
5:                 a[j] ↔ a[j + 1]
6:             end if
7:         end for
8:     end for
9:     return a[1..n]
10: end function
```

Insertion Sort

Idee: Insertion Sort, vezi *Algorithm 2*, consideră lista care urmeaza a fi sortată ca fiind împărțită în două părți, una sortată (în partea stângă) și una nesortată (în partea dreaptă). Algoritmul începe sortarea de la al doilea element, ducând fiecare element pe poziția potrivită în subșirul sortat, astfel încât subșirul din partea stangă să rămână sortat.

Pentru a fi mai uşor de imaginat modul în care funcţionează Insertion Sort, putem să ne imaginăm modul în care amestecăm un pachet de cărţi sau pentru cititorii împotriva jocurilor cu cărţi putem considera modul în care oamenii se aranjează în ordine crescătoare sau descrescătoare, după înălţime, într-o coloană. Aceştia îşi compară înălţimea cu a celui curent, iar în funcţie de rezultat merg mai departe cu următorul om din coloană sau se opresc în locul respectiv.

La fel ca în cazul Bubble Sort, nu putem şti cine a inventat acest algoritm. Se consideră că acest algoritm a fost folosit cu mult timp înainte de inventarea calculatoarelor sau a termenului de algoritm. El este în esenţă un algoritm "comun", reprezentând modul simplu în care oamenii sortează o listă de elemente.

Pentru a analiza timpul de executie al acestui algoritm este nevoie să determinăm operaţia dominantă. În acest caz vom considera cele două comparaţii din algoritm $j > 0$ and $aux < a[j]$. Desigur, pentru acest algoritm trebuie să identificăm cazul favorabil si cel nefavorabil. Cazul favorabil este atunci când tabloul unidimensional este deja sortat, caz în care avem:

$$T(n) = \sum_{i=2}^{n} 1 = n - 1 \approx n \Rightarrow T(n) \in \Omega(n)$$

Cazul nefavorabil este atunci când avem tabloul unidimensional sortat pe dos, caz în care avem:

$$T(n) = \sum_{i=2}^{n} \sum_{j=1}^{n} 1 = \sum_{i=2}^{n} n = n(n-1) = n^2 - n \approx n^2 \Rightarrow T(n) \in O(n^2)$$

După cum putem observa, algoritmul are o creştere liniară în cazul favorabil şi o creştere pătratică în cazul nefavorabil. Intuitiv, putem spune că acest algoritm este util doar dacă este folosit în cazul sortării unei liste aproape sortate.

Algorithm 2 Insertion Sort

```
1: function SORT(a[1..n])
2:     for i ← 2, n do
3:         aux ← a[i]
4:         j ← i − 1
5:         while j > 0 and aux < a[j] do
6:             a[j + 1] ← a[j]
7:             j ← j − 1
8:         end while
9:         a[j + 1] ← aux
10:    end for
11:    return a[1..n]
12: end function
```

Selection Sort

Idee: Selection Sort, vezi *Algorithm 3*, asemenea Insertion Sort, consideră lista care urmeaza a fi sortată ca fiind împărțită în două părți, una sortată (în partea stângă) și una nesortată (în partea dreaptă). Inițial, partea sortată este vidă, iar partea nesortată conține toate elementele listei. Algoritmul caută în partea nesortată minimul (considerând sortarea crescătoare a elementelor) pe care îl interschimbă cu primul element aflat după partea sortată, astfel încât subșirul din partea stângă să rămână sortat. O imagine din lumea reală ar fi modul în care oameii vor să aleagă doar primele k obiecte după mărime. Mai întâi trebuie să aleagă primul cel mai mare element (dacă presupunem sortarea descrescătoare), apoi următorul cel mai mare element şi tot aşa până când se ajunge la k elemente selectate. Singura diferenţă este că nu vorbim de k elemente din cele n, ci de toate cele n.

La fel ca în cazul Bubble Sort, nici Selection Sort nu are caz favorabil

şi nefavorabil sau, altfel spus, cele două coincid. Dacă alegem ca operaţie dominantă comparaţia $a[i] < a[min]$ atunci vom obţine următorul timp de execuţie:

$$T(n) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} 1 = \sum_{i=1}^{n} (n-i) = n-1+n-2+n-3+...+n-n+1+n-n =$$

$$= 1 + 2 + ... + n - 2 + n - 1 = \frac{(n-1)n}{2} = \frac{n^2 - n}{2} \approx n^2 \Rightarrow T(n) \in \Theta(n)$$

Algorithm 3 Selection Sort

```
1: function SORT(a[1..n])
2:     for i ← 1, n do
3:         min ← i
4:         for j ← i + 1, n do
5:             if a[j] < a[min] then
6:                 min ← j
7:             end if
8:         end for
9:         if min ≠ i then
10:            a[min] ↔ a[i]
11:        end if
12:    end for
13:    return a[1..n]
14: end function
```

Quick Sort

Idee: Quick Sort, vezi *Algorithm 5*, este un algoritm bazat pe Divide et Impera. A fost inventat de informaticianul britanic Tony Hoare în 1959 şi face parte din lista algoritmilor clasici de sortare. Quick Sort împarte lista de sortat în două liste mai uşor de sortat. Acesta alege un element din listă, denumit **pivot**, vezi *Algorithm 4*, și rearanjează lista, astfel încât pivotul se află pe poziția sa finală, iar toate elementele din stânga sa sunt mai mici decât el și toate elementele din dreapta sa sunt mai mari decât el (considerând sortarea crescătoare). Pivotul este ales total aleator. Desigur, există variante ale algoritmului care aleg pivotul în funcţie de elementele din listă, astfel că poziția finală a sa este aproximativ în mijlocul listei. Noi vom alege varianta simplă în care elementul din dreapta listei este selectat drept pivot. După această partiționare, algoritmul aplică aceeași pași și pe sublistele

aflate în stânga pivotului, respectiv în dreapta lui. Algoritmul se aplică cât timp există cel puţin două elemente în sublistă.

Spre deosebire de ceilalţi algoritmi de până acum, Quick Sort este constituit din două funcţii, una care face partiționarea şi pune elementul pivot pe poziția sa finală şi una care apelează funcţia de partiţionare şi imparte tabloul curent în două subtablouri pentru aplicarea aceluiaşi procedeu. Altfel fiind spus, va trebui să luăm in considerare ambele funcţii pentru determinarea timpului de execuţie. Prima dată vom începe cu funcţia de partiţionare. Drept operaţie dominantă vom alege comparaţia $a[j] \leq pivot$. Astfel, timpul de execuţie va fi

$$T(n) = \sum_{j=s}^{d-1} 1 = d - 1 - s + 1 = d - s = n - 1 \approx n \Rightarrow T(n) \in \Theta(n)$$

Indiferent de datele de intrare, partiţionarea va aparţine lui $\Theta(n)$. În schimb, pentru funcţia de QuickSort va trebui să luăm în calcul cazul favorabil şi cazul nefavorabil. Cazul favorabil este atunci când în urma partiţionării obţinem poziţia pivotului ca fiind mijlocul tabloului de partiţionat. Pe langă acestea, nu putem exclude din calcul nici apelul funcţiei de partiţionare. Astfel, ajungem la următoarea relaţie de recurenţă:

$$T(n) = \begin{cases} 0 & , n \leq 1 \\ 2T(\frac{n}{2}) + n & , n > 1 \end{cases}$$

Pentru a calcula timpul de execuţie mai uşor vom folosi o teoremă pentru rezolvarea recurenţei, numită Teorema Master. Pentru mai multe detalii referitoare la Teorema Master, consultă cartea "Introduction to Algorithms" de Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, secţiunea 4.5 (The master method for solving recurrences), pag. 93. Deoarece avem $a = 2$, $b = 2$ şi $d = 1$, rezultă din cazul al doilea al Teoremei Master că $T(n) \in \Omega(n \log n)$.

Cazul nefavorabil este atunci când tabloul este împărţit într-un tablou cu $n - 1$ elemente şi un tablou vid. Altfel spus, cazul nefavorabil este atunci când tabloul este sortat, indiferent de sortare. Pentru acest caz funcţia de partiţionare va arăta la fel, după cum am precizat mai sus, însă funcţia de QuickSort va arăta astfel:

$$T(n) = \begin{cases} 0 & , n \leq 1 \\ n + T(n-1) + T(0) = n + T(n-1) & , n > 1 \end{cases}$$

Pentru a putea rezolva formula recurentă vom rescrie formula pentru numere de la n până la 1, fiecare formulă una sub cealaltă. Astfel, vom obţine formulele:

$$\begin{array}{lcl} T(n) & = & T(n-1)+n \\ T(n-1) & = & T(n-2)+n-1 \\ ... & ... & ... \\ T(2) & = & T(1)+2 \\ T(1) & = & 0 \end{array}$$

După ce adunăm toate formulele şi simplificăm ce putem, vom obţine:

$$T(n) = n+(n-1)+...+2+1-1 = \frac{n(n+1)}{2}-1 = \frac{n^2+n}{2}-1 \approx n^2 \Rightarrow T(n) \in O(n^2)$$

După cum am putut observa, Quick Sort are o creştere liniară în cazul favorabil şi o creştere pătratică în cazul nefavorabil. În realitate, Quick Sort este mai apropiat de cazul favorabil decât de cel nefavorabil, deoarece $T_{mediu} \approx n \log n$. Acest lucru îl plasează în lista algoritmilor eficienţi şi destul de uşor de implementat, cel puţin în această variantă.

Algorithm 4 Partiţionare

```
1: function PARTITION(a[s..d])
2:     pivot ← a[d]
3:     i ← s − 1
4:     for j ← s, d − 1 do
5:         if a[j] ≤ pivot then
6:             i ← i + 1
7:             a[i] ↔ a[j]
8:         end if
9:     end for
10:    a[i + 1] ↔ a[d]
11:    return i + 1
12: end function
```

Algorithm 5 QuickSort

```
1: procedure QUICKSORT(a[s..d])
2:     if s < d then
3:         q = partition(a[s..d])
4:         quicksort(a[s..q − 1])
5:         quicksort(a[q + 1..d])
6:     end if
7: end procedure
```

Merge Sort

Idee: Merge Sort, vezi *Algorithm 7*, este un algoritm bazat pe Divide et Impera, inventat de informaticianul John von Neumann în anul 1945. Ideea algoritmului este destul de simplă. Acesta împarte lista de sortat în două subliste de lungimi aproximativ egale. Algoritmul împarte fiecare sublistă, recursiv, folosind aceeași pași, după care le interclasează, vezi *Algorithm 6*, pentru a obține lista inițială sortată.

Pentru a putea vizualiza mai uşor modul în care Merge Sort funcţionează putem să ne gândim că suntem un grup de oameni şi vrem să sortăm un număr foarte mare de obiecte după dimensiune. Pentru a termina mai repede putem să împărţim grămada de obiecte aleator între noi, în grămezi egale, iar apoi fiecare dintre noi să-şi sorteze grămada. După ce am terminat de sortat putem să ne grupăm doi câte doi şi să formăm din cele două grămezi sortate o singură grămadă sortată, după care repetăm acest procedeu până când se formează o singură grămadă care conţine toate obiectele grămezii iniţiale. De fapt, am obţinut grămada iniţială sortată.

La fel ca în cazul Quick Sort, este necesară analiza fiecărei funcţii. Vom începe cu analiza funcţiei de interclasare. Pentru aceasta vom alege drept operaţie dominantă incrementarea $k \leftarrow k + 1$. Desigur, puteam alege şi comparaţiile, însă am ales această variantă pentru a fi mai uşor de urmărit. Drept urmare, timpul de execuţie este

$$T(n) = \sum_{i=s}^{m} 1 + \sum_{j=m+1}^{d} 1 = m-s+1+d-m-1+1 = d-s+1 = n \Rightarrow T(n) \in \Theta(n)$$

În scrierea sumelor am luat în calcul faptul că dintre ultimele două cicluri *while* se va intra doar într-unul din ele, cel în care se vor adăuga elemente din partea de tablou rămasă neparcursă. Deci, interclasarea este de complexitate liniară întotdeauna. De altfel, acest rezultat este de aşteptat ţinând cont că nu facem altceva decât să parcurgem tot tabloul, dar din două parţi, pe rând, până când punem toate elementele sortate în tablou. Pentru calculul timpului de execuţie al funcţiei *mergesort* va trebui să scriem din nou relaţia de recurenţă, ea fiind o funcţie recurentă.

$$T(n) = \begin{cases} 0 & , n = 1 \\ T([n/2]) + T(n - [n/2]) + n & , n > 1 \end{cases}$$

Pentru calculul timpului de execuţie avem $a = 2$, $b = 2$ şi d=1, astfel că vom folosi din nou cazul al doilea din Teorema Master şi vom obţine $T(n) \in \Theta(n \log n)$. De această dată nu avem caz favorabil şi caz nefavorabil, deoarece

algoritmul împarte de la început tabloul în două tablouri aproape egale. Cu toate acestea există, totuşi, un dezavantaj. Algoritmul are nevoie de memorie adiţională, lucru care poate fi uneori neplăcut. Chiar dacă se câştigă timp, se pierde memorie. Întotdeauna trebuie făcut un compromis.

Algorithm 6 Interclasare

```
1: function MERGE(a[s..m], b[m + 1..d])
2:     c[1..d − s + 1]                                  ▷ tablou unidimensional
3:     k ← 0
4:     i ← s; j ← m + 1
5:     while i ≤ m and j ≤ d do
6:         k ← k + 1
7:         if a[i] ≤ a[j] then
8:             c[k] ← a[i]
9:             i ← i + 1
10:        else
11:            c[k] ← a[j]
12:            j ← j + 1
13:        end if
14:    end while
15:    while i ≤ m do
16:        k ← k + 1
17:        c[k] ← a[i]
18:        i ← i + 1
19:    end while
20:    while j ≤ d do
21:        k ← k + 1
22:        c[k] ← a[j]
23:        j ← j + 1
24:    end while
25:    return c[1..k]                                   ▷ k=m+n
26: end function
```

Algorithm 7 Merge Sort

```
1: function MERGESORT(a[s..d])
2:     if s < d then
3:         m ← (s + d)/2
4:         a[s..m] ← mergesort(a[s..m])
5:         a[m + 1..d] ← mergesort(a[m + 1..d])
6:         a[s..d] ← merge(a[s..m], a[m + 1..d])
7:     end if
8:     return a[s..d]              ▷ În cazul în care s = d, funcţia va returna
9:                                    ▷ un tablou cu un singur element
10: end function
```

3 Analiza experimentală a algoritmilor elementari de sortare

Pentru implementarea algoritmilor am folosit limbajul de programare Python, versiunea 3.7 a acestuia. Laptopul utilizat este un Samsung Series 7 Chronos NP770Z5C-S02RO cu procesor Intel Core i5-3210M pe 64 de biți, 2.50 GHz și 6 GB memorie RAM. Pentru fiecare algoritm laptopul a fost pus pe modul "High Performance" și conectat la priză, iar rularea s-a efectuat în timp ce toate celelalte programe au fost închise pentru a avea parte de o analiză experimentală cât mai corectă.Pe scurt, toți algoritmii au fost testați în aceleași condiții și tratați în același mod.

Ca să înţelegem mai bine ce concluzii să tragem din experiment, propun să analizăm puţin datele de intrare. Pentru acest experiment am propus trei tipuri de date de intrare. Primul tip de date de intrare se referă la datele generate aleator. Elementele din primul tip sunt elemente ce aparţin mulţimii $[1, 50000]$. Fără nici cea mai mică strategie am ales acest interval, ci doar din plăcere. Pentru cel de-al doilea tip de date de intrare am ales cazul nefavorabil în care elementele sunt sortate în ordine inversă. În particular, am ales tipul al doilea de date constituit din elemente sortate descrescător, deoarece noi ne dorim să obţinem la final tabloul ordonat crescător. Elementele sunt de la n la 1 în această ordine, n fiind numărul de elemente. Tipul al treilea de date se referă la un set de numere de la 1 la n, în aceasta ordine, la care o zecime din elemente au fost interschimbate. Astfel, am obţinut un şir aproape sortat. Acest ultim tip de date ne va arăta comportamentul lui Insertion Sort pe tablouri aproape sortate. Pentru fiecare algoritm şi tip de date de intrare am început generarea a 100 de elemente. Treptat, am crescut numărul de elemente generate până când timpul de execuţie a devenit prea mare şi, deci, algoritmul ineficient. Înainte să începem analiza rezultatelor,

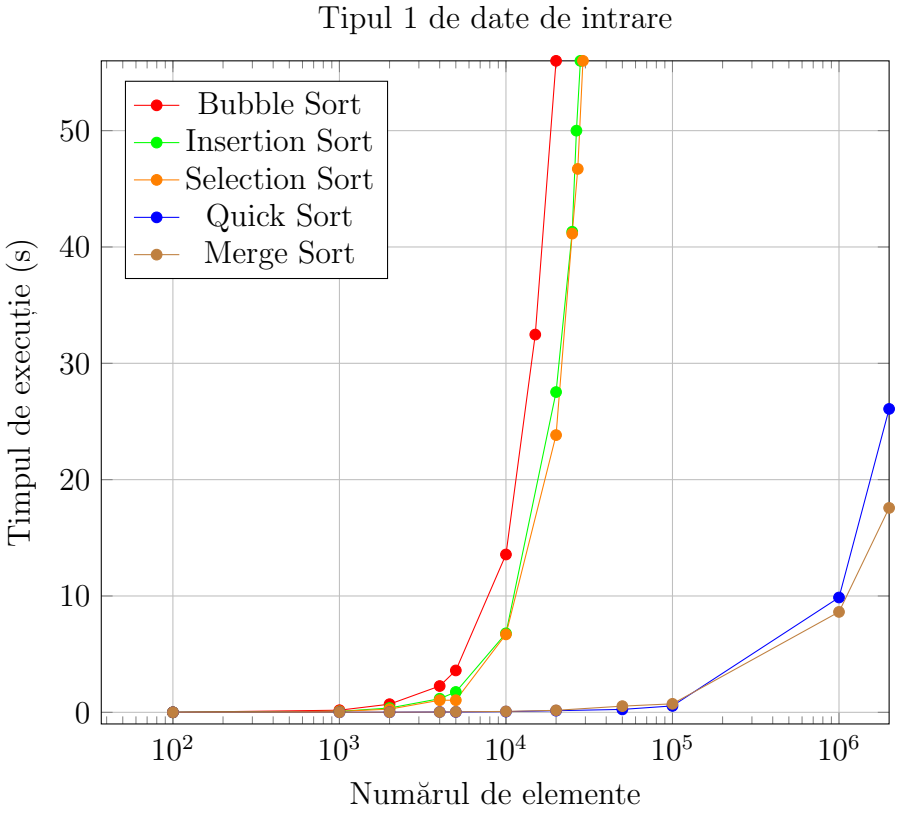
menţionez faptul că există şanse destul de mari ca implementarea recursivă a Quick Sort să genereze o eroare "*RuntimeError: maximum recursion depth exceeded*". Acest lucru se datorează din cauza limitării implicite a lui Python, limitare de 100 de nivele pe stivă. Există mai multe moduri de a remedia această problemă. Totuşi, două sunt destul de cunoscute şi au efect de sută la sută. Una dintre ele este să setăm noi limita nivelelor de pe stivă. Cu toate că este foarte simplu de făcut acest lucru, nu este recomandat, deoarece se poate întâmpla să stricăm stiva şi, deci, să stricăm mai mult decât reparăm. A doua şi cea mai puţin riscantă varianta este să implementăm varianta iterativă a lui Quick Sort. Aceasta nu este foarte complicată sau diferită de cea recursivă. Funcţia de partiţionare rămâne aceeaşi. Singura deosebire este la funcţia *quicksort*, unde va trebui să avem noi o stivă explicită pentru a simula stiva de la apelul recursiv. Mai jos se află o imagine cu codul aferent.

```
def quick_sort(x,s,d):

    lungimea=d-s+1
    stack=[0]*lungimea
    varf=-1 #init
    varf+=1 #push
    stack[varf]=s
    varf+=1 #push
    stack[varf]=d

    while varf>=0:
        dreapta=stack[varf]
        varf-=1 #pop
        stanga=stack[varf]
        varf-=1 #pop
        p=partitionare(x, stanga, dreapta)
        if p-1>stanga:
            varf+=1
            stack[varf]=stanga
            varf+=1
            stack[varf]=p-1
        if p+1<dreapta:
            varf+=1
            stack[varf]=p+1
            varf+=1
            stack[varf]=dreapta
```

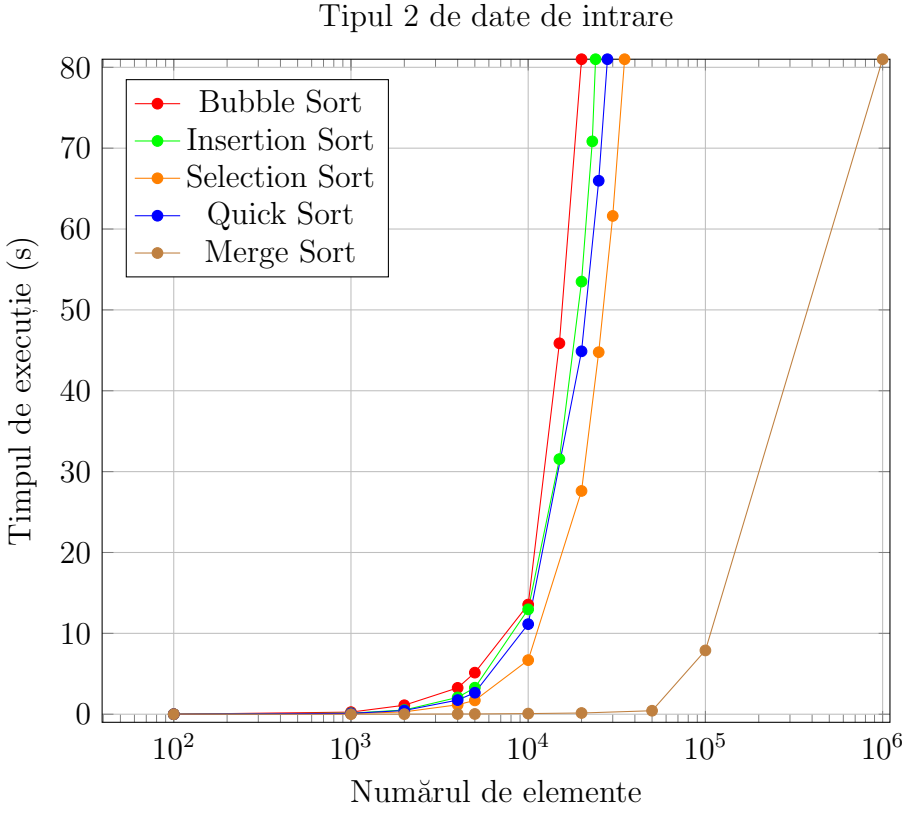
Figura 1: Implementarea iterativă a lui Quick Sort.

Acum că am analizat datele de intrare, putem să trecem la analiza timpului de execuţie. Pentru calculul timpului de execuţie am folosit metoda *time* din pachetul *time*. Apelăm metoda înainte de sortarea tabloului şi după sortarea tabloului, reţinând rezultatele. La final, ne rămâne doar să facem

diferenţa în modul dintre acestea şi aşa am obţinut timpul de execuţie. Pentru a fi mai uşor de citit, vom pune rezultatele într-un grafic.

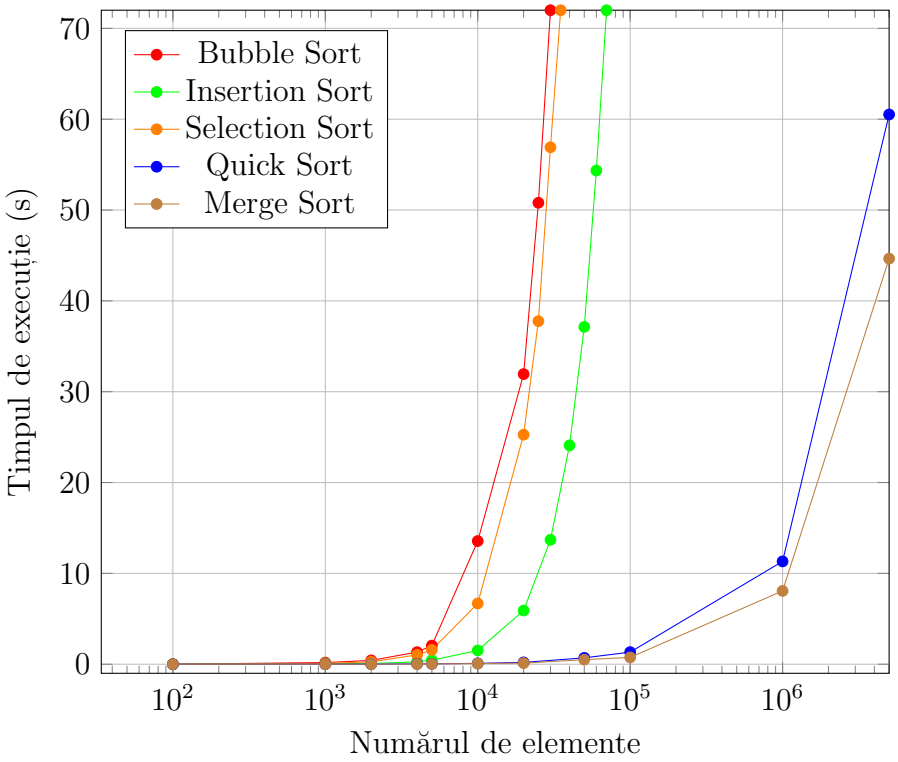


Din primul grafic putem observa că algoritmul câştigător este Merge Sort, urmat de Quick Sort. Acesta din urmă se distanţează destul de bine când vine vorba de minim 10^6 de elemente. În cealaltă parte, la coada şirului se afla Bubble Sort. De altfel, acest rezultat era de aşteptat pentru ambele părţi, cea câştigătoare şi cea învinsă, având în vedere complexităţiile algoritmilor. Puţin surprinzător este Selection Sort care foarte puţin reuşeşte să întreacă Insertion Sort. Acest lucru se poate datora faptului că Selection Sort nu face atâtea mutări sau interschimbări precum Insertion Sort. Deci, pentru cazul în care avem date în ordine aleatoare putem folosi Merge Sort pentru a sorta mai rapid datele.



În al doilea grafic am reuşit să surprindem modul în care algoritmii reacţionează în cazul nefavorabil al sortării. În mod deloc surprinzător, Quick Sort se situează printre Selection Sort, Bubble Sort şi Insertion Sort. Conform analizei teoretice acest lucru este posibil, Quick Sort având o complexitate $O(n^2)$. Din nou putem observa cum din cei patru, Selection Sort tinde să fie mai rapid. Exact ca şi mai sus, acest lucru se datorează faptului că Selection Sort nu face atât de multe interschimbări. Aşa cum ne aşteptam, Merge Sort este câştigător, el având o complexitate $\Theta(n \log n)$. Până acum nu este nimic neobişnuit în comportamentul algoritmilor.

Tipul 3 de date de intrare



Pentru ultimul grafic putem observa din nou partea stângă formată din Bubble Sort, Insertion Sort şi Selection Sort şi partea dreaptă formată din Quick Sort şi Merge Sort. De data aceasta, faţă de primele două grafice, avem o mare diferenţă în partea stângă. Este vizibil cu ochiul liber că în partea stângă putem desemna drept câştigător pe Insertion Sort. Acesta se detaşează de ceilalţi algoritmi ai aceleiaşi părţi mai tare decât s-a detaşat Selection Sort în primele două grafice. Acest comportament este, de asemenea, unul aşteptat, ştiind din analiza teoretică că Insertion Sort este mai eficient pe tablouri aproape sortate decât Bubble Sort şi Selection Sort. Cu toate acestea, tot Merge Sort iese învingător. De această dată acesta se detaşează mai repede de Quick Sort, începând de la 10^5. Un alt comportament normal, ţinând cont că orice tablou sortat sau mai mult sortat decât nesortat ne conduce spre cazul nefavorabil al lui Quick Sort.

4 Concluzii

Algoritmii de sortare sunt vitali în viaţa de zi cu zi şi sunt cu atât mai importanţi cu cât datele devin mai numeroase. În fiecare zi ne folosim de

sortări fără să ne dăm seama, de la statul într-un şir descrescător la sport la căutatul mai rapid a unei persoane din agendă prin simplul fapt că persoanele din agendă sunt sortate alfabetic. Acestea fiind spuse, nu putem ignora algoritmii de sortare sau să-i refuzăm în viaţa noastră. Totodată, luând în calcul analizele şi observaţiile făcute, este important să cunoaştem bine problema şi datele de intrare, deoarece s-ar putea să putem folosi mai eficient un algoritm decât un altul. Din ceea ce am văzut este bine să reţinem faptul că Bubble Sort este de cele mai multe ori ineficient, Selection Sort poate învinge în multe cazuri Bubble Sort şi Insertion Sort, Merge Sort este cel mai rapid dintre cei cinci, însă are nevoie de memorie adiţională, lucru care poate fi puţin neplăcut, Quick Sort se poate alătura lui Bubble Sort, Selection Sort şi Insertion Sort când este în cazul său nefavorabil, iar Insertion Sort este destul de eficient pe liste aproape sortate. Deci, fiecare are avantajele şi dezavantajele lui. Important este să ştim la ce putem renunţa pentru a avea altceva în plus.

Bibliografie

[1] Granville Barnett, Luca Del Tongo . *Data Structures and Algorithms:Annotated Reference with Examples First Edition*. DotNetSlackers, 2008.

[2] Steven S. Skiena. *The Algorithm Design Manual Second Edition*. Springer, 2008.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms Third Edition*. Cambridge, Massachusetts, 2009.

[4] Mohsin Khan, Samina Shaheen, Furqan Aziz Qureshi. *Comparative Analysis of five Sorting Algorithms on the basis of Best Case, Average Case, and Worst Case*. International Journal of Information Technology and Electrical Engineering, 2012-2013.

[5] Robert Sedgewick, Kevin Wayne. *Algorithms Fourth Edition*. Pearson Education, 2011.

[6] Donald Knuth. *The Art of Computer Programming Third Edition*. Addison-Wesley, 1938.

[7] Owen Astrachan. *Bubble Sort: An Archaeological Algorithmic Analysis*. ACM, 2003.

[8] Jeffrey J. McConnell. *Analysis of Algorithms: An Active Learning Approach*. ones and Bartlett, 2001.