Metode de sortare -comparație

Iulian-Iosif Poenaru

2023

Lucracrea cuprinde următoarele sortări:

* Sortarea prin selecție
* Sortarea “Bubble sort”
* Sortarea prin inserție
* Sortarea prin interclasare
* Sortarea “Quick sort”
* Sortarea prin numărare
* Sortarea „Cocktail sort”

Observație:

* Toate sortările au fost făcute astfel încât șirul să fie sortat crescător.
* Algoritmii de sortare au fost executați într-un singur limbaj de programare și anume Python. Există posibilitatea ca timpii de execuție să fie mai mici pentru un limbaj ce nu este de nivel înalt (exemplu C,C++).
* În unele cazuri pentru siruri de lungime mai mari ca 104 nu s-a făcut media între 3 teste, media reprezentând primul test.

**Sortarea prin selecție**

Algoritmul se bazează pe constanta aducere a minimului din zona nesortată a șirului pe prima poziție a zonei nesortate. Exemplu:

În cazul de față zona nesortată este reprezentată de întreg șirul, astfel se dorește aducerea minimului pe prima poziție

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 64 | 25 | 12 | 22 | 11 | 4 |

Minimul șirului este pe prima poziție. Acum procesul anterior se repetă pentru noua zonă nesortată

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 4 | 25 | 12 | 22 | 11 | 64 |

Procesul se repetă până când întreg șirul este sortat.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 4 | 11 | 12 | 22 | 25 | 64 |

Timpul de executie: *T(n) = n2*

Ordinul de complexitate: *O(n2)*

Spațiu de memorie: *O(1)*

**Date experimentale:**

*CAZ1* Pentru un sir cu elemente nesortate:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nr de elemente | T(n) în secunde | | | |
|  | Test1 | Test2 | Test3 | Media |
| 10 | 0.0 | 0.0 | 0.0 | 0.0 |
| 100 | 0.001997 | 0.001993 | 0.002991 | 0.002327 |
| 1000 | 0.137662 | 0.157577 | 0.143642 | 0.146293 |
| 10\_000 | 13.774364 | 12.848693 | 13.647022 | 13.423359 |
| 100\_000 | 1936.683068 | - | - | 1936.683068 |
| 1\_000\_000 | >30min | - | - | >30min |
| 100\_000\_000 | >30min | - | - | >30min |

*CAZ2* Pentru un sir cu elemente sortate crescator aproape complet\*:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nr de elemente | T(n) în secunde | | | |
|  | Test1 | Test2 | Test3 | Media |
| 10 | 0.0 | 0.0 | 0.0 | 0.0 |
| 100 | 0.000959 | 0.001994 | 0.001001 | 0.001318 |
| 1000 | 0.000996 | 0.000996 | 0.000998 | 0.000996 |
| 10\_000 | 14.515559 | 14.320239 | 13.750690 | 14.195496 |
| 100\_000 | 1991.765871 | - | - | 1991.765871 |
| 1\_000\_000 | >30min | - | - | >30min |
| 100\_000\_000 | >30min | - | - | >30min |

\* n-n/4 elemente sortate și restul nesortate

*CAZ3*  Pentru un sir cu elementele sortate descrescator:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nr de elemente | T(n) în secunde | | | |
|  | Test1 | Test2 | Test3 | Media |
| 10 | 0.0 | 0.0 | 0.0 | 0.0 |
| 100 | 0.001026 | 0.000997 | 0.001996 | 0.001339 |
| 1000 | 0.140624 | 0.168550 | 0.150597 | 0.153257 |
| 10\_000 | 15.883891 | 15.381237 | 15.841245 | 15.702124 |
| 100\_000 | 2540.836802 | - | - | 2540.836802 |
| 1\_000\_000 | >30min | - | - | >30min |
| 100\_000\_000 | >30min | - | - | >30min |

Număr de execuții:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nr de execuții op dominantă | | Nr de intrări pentru valoarea de adevăr A\* | | |
| Teoretic (n2) | Practic | Caz 1 | Caz 2 | Caz 3 |
| 100 | 45 | 14 | 1 | 25 |
| 10000 | 4950 | 279 | 49 | 2500 |
| 1000000 | 499.500 | 4966 | 995 | 250\_000 |
| 1000000000 | 49995000 | 74\_697 | 15\_505 | 25\_000\_000 |
| 1000002 | 4999950000 | 994\_804 | 210\_135 | 2\_500\_000\_000 |
| 10000002 | - | - | - | - |
| 10000000002 | - | - | - | - |

\* algoritmul se bazează pe compararea elemntelor folosind instrucțiunea dacă(if)

**Concluzie:** Algoritmul este destul de eficient pe seturi mici de date, dar este ineficient pe seturi mari >103. Primele trei tabele reprezinta trei cazuri abordate. Primul caz este situația cea mai comună, situație în care șirul cuprinde elemente care nu se află înr-o ordine anume. În al doilea caz elementele sunt semisortate (aproape sortate), fapt ce crește eficiența algoritmului, iar ultimul caz și cel în care algoritmul are cel mai mult de lucru este când elementele sunt sortate descrescător. Ultimul tabel are scopul de a evidenția faptul că algoritmul este cel mai eficient atunci când elementele sunt semisortate crescător. De asemenea, observăm că ordinul de execuție dat de repetiția operației dominante nu este exact același cu cel din teorie, fapt ce se datorează restrângerii zonei de căutare după fiecare interschibare de elemnte.

**Sortarea “Bubble Sort”**

Algoritmul se bazează pe interschimbarea elementelor vecine dacă aceastea nu se află în ordinea corespunzătoare. Exemplu:

Se verifică dacă cele două elemente sunt în ordine și se face interschimbarea dacă nu se află în ordine.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 64 | 25 | 12 | 22 | 11 | 4 |

Se continuă pasul anterior până se ajunge pe ultima poziție.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 25 | 64 | 12 | 22 | 11 | 4 |

Apoi se repetă pașii de la început până când algoritmul este sortat.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 25 | 12 | 22 | 11 | 4 | 64 |

Timpul de executie: *T(n) = n2*

Ordinul de complexitate: *O(n2)*

Spațiu de memorie: *O(1)*

**Date experimentale:**

Pentru un sir cu elemente nesortate:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nr de elemente | T(n) în secunde | | | |
|  | Test1 | Test2 | Test3 | Media |
| 10 | 0.0 | 0.0 | 0.0 | 0.0 |
| 100 | 0.001964 | 0.001999 | 0.001994 | 0.001985 |
| 1000 | 0.196478 | 0.194482 | 0.197327 | 0.196095 |
| 10\_000 | 20.042912 | 19.762274 | 21.388425 | 20.846374 |
| 100\_000 | 3056.7944 | - | - | 3056.7944 |
| 1\_000\_000 | >30min | - | - | >30min |
| 100000000 | >30min | - | - | >30min |

Pentru un sir cu elemente sortate crescator aproape complet\*:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nr de elemente | T(n) în secunde | | | |
|  | Test1 | Test2 | Test3 | Media |
| 10 | 0.0 | 0.0 | 0.0 | 0.0 |
| 100 | 0.000997 | 0.000996 | 0.001998 | 0.000998 |
| 1000 | 0.001993 | 0.000997 | 0.001003 | 0.001331 |
| 10\_000 | 15.184555 | 15.309383 | 14.631397 | 15.041778 |
| 100\_000 | 2099.468450 | - | - | 2099.468450 |
| 1\_000\_000 | >30min | - | - | >30min |
| 100\_000\_000 | >30min | - | - | >30min |

\* n-n/4 elemente sortate și restul nesortate

Pentru un sir cu elementele sortate descrescator:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nr de elemente | T(n) în secunde | | | |
|  | Test1 | Test2 | Test3 | Media |
| 10 | 0.0 | 0.0 | 0.0 | 0.0 |
| 100 | 0.002991 | 0.003988 | 0.002991 | 0.003323333 |
| 1000 | 0.002961 | 0.004015 | 0.002993 | 0.003323000 |
| 10\_000 | 29.665394 | 30.619866 | 30.950322 | 30.411860 |
| 100\_000 | 4459.526425 | - | - | 4459.526425 |
| 1\_000\_000 | >30min | - | - | >30min |
| 100\_000\_000 | >30min | - | - | >30min |

Număr de execuții:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nr de execuții op dominantă | | Nr de intrări pentru valoarea de adevăr A\* | | |
| Teoretic (n2) | Practic | Caz 1 | Caz 2 | Caz 3 |
| 100 | 54 | 27 | 0 | 45 |
| 10000 | 5049 | 2025 | 125 | 4950 |
| 1000000 | 500499 | 178134 | 11546 | 499500 |
| 1000000000 | 50004999 | 18\_392\_835 | 1\_127\_643 | 49\_995\_000 |
| 1000002 | 5000049999 | 1\_837\_381\_268 | 115\_704\_988 | 4\_999\_950\_000 |
| 10000002 | - | - | - | - |
| 1000000002 | - | - | - | - |

\* algoritmul se bazează pe compararea elemntelor folosind instrucțiunea dacă(if)

**Concluzie:** Algoritmul este eficient pentru seturi mici de date, iar și mai eficient când lucrezi cu date care sunt aproape sortate. Comparând cazurile 2 și 3(tabelul 2 și 3) observăm diferența mare dintre timpii de execuție, iar analizând tabelul 4 observăm importanța input-ului. Cu cât acesta este mai aproape de versiunea ideală cu atât algoritmul se descurcă mai bine. De asemenea, observăm că ordinul de execuție dat de repetiția operației dominante nu este exact același cu cel din teorie, fapt ce se datorează restrângerii zonei de căutare după fiecare interschibare de elemnte.

**Sortarea prin insertie**

Algoritmul verifica constant elementele vecine si le interschimba in cazul în care acestea nu se află în ordinea corectă.

[](https://media.geeksforgeeks.org/wp-content/uploads/insertionsort.png)

Timpul de executie: *T(n) = n2*

Ordinul de complexitate: *O(n2)*

Spațiu de memorie: *O(1)*

**Date experimentale:**

Pentru un sir cu elemente nesortate:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nr de elemente | T(n) în secunde | | | |
|  | Test1 | Test2 | Test3 | Media |
| 10 | 0.0 | 0.0 | 0.0 | 0.0 |
| 100 | 0.001993 | 0.000997 | 0.001021 | 0.001337 |
| 1000 | 0.131617 | 0.127660 | 0.118872 | 0.126049 |
| 10\_000 | 12.641489 | 12.749808 | 13.454897 | 12.948731 |
| 100\_000 | 1430.627303 | - | - | 1430.627303 |
| 1\_000\_000 | >30min | - | - | >30min |
| 100000000 | >30min | - | - | >30min |

Pentru un sir cu elemente sortate crescator aproape complet\*:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nr de elemente | T(n) în secunde | | | |
|  | Test1 | Test2 | Test3 | Media |
| 10 | 0.0 | 0.0 | 0.0 | 0.0 |
| 100 | 0.0 | 0.0 | 0.0 | 0.0 |
| 1000 | 0.010004 | 0.009999 | 0.012964 | 0.010989 |
| 10\_000 | 0.818143 | 0.806351 | 0.811449 | 0.811981 |
| 100\_000 | 90.292905 | 88.476657 | 81.875330 | 86.881630 |
| 1\_000\_000 | >60min | - | - | >60min |
| 100\_000\_000 | >60min | - | - | >60min |

\* n-n/4 elemente sortate și restul nesortate

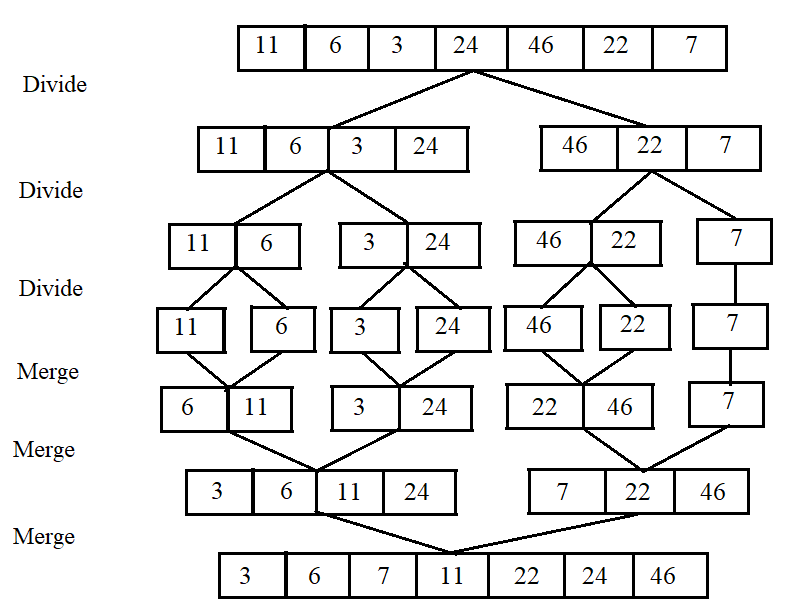
Pentru un sir cu elementele sortate descrescator:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nr de elemente | T(n) în secunde | | | |
|  | Test1 | Test2 | Test3 | Media |
| 10 | 0.0 | 0.0 | 0.0 | 0.0 |
| 100 | 0.001990 | 0.002023 | 0.001994 | 0.006007 |
| 1000 | 0.259690 | 0.268452 | 0.276812 | 0.268318 |
| 10\_000 | 25.903755 | 29.065221 | 25.390089 | 26.786355 |
| 100\_000 | 2650.222237 | - | - | 2650.222237 |
| 1\_000\_000 | >30min | - | - | >30min |
| 100\_000\_000 | >30min | - | - | >30min |

**Concluzie:** Algoritmul este eficient pentru seturi mici de date, dar nu se descurcă rău nici pe seturi mai mari de date. Comparativ cu algoritmii anteriori, sortarea prin inserție are timpi de execuție foarte buni în momentul în care lucrăm cu șiruri aproape sortate, dar în momentul în care șirul este deja sortat descrescător, algoritmul scade în eficiență. Atât pentru seturi mic de date, cât și pentru seturi mai mari se poate observa diferența asupra timpului de execuție atunci când algoritmul este aproape sortat crescător (pentru un input cu elemente aranjate aleator algoritmul are nevoie de aproximativ jumătate de oră, iar pentru un sir aproape sortat are nevoie de aproximativ 1 minut jumătate).

**Sortarea prin interclasare**

Principiul de bază al algoritmului este reducerea zonei de sortare astfel:

[](https://cdn.educba.com/academy/wp-content/uploads/2021/06/7.png)

Timpul de executie: *T(n) = nlogn*

Ordinul de complexitate: *O(nlogn)*

Spațiu de memorie: *O(n)*

**Date experimentale:**

Pentru un sir cu elemente nesortate:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nr de elemente | T(n) în secunde | | | |
|  | Test1 | Test2 | Test3 | Media |
| 10 | 0.0 | 0.0 | 0.0 | 0.0 |
| 100 | 0.000966 | 0.000996 | 0.000996 | 0.000996 |
| 1000 | 0.008005 | 0.008974 | 0.009945 | 0.008974 |
| 10\_000 | 0.127659 | 0.118683 | 0.121773 | 0.122705 |
| 100\_000 | 1.482950 | 1.364557 | 1.399966 | 1.415824 |
| 1\_000\_000 | 16.728478 | 16.571618 | 16.626714 | 16.64227 |
| 100\_000\_000 | 2222.685373 | - | - | 2222.685373 |

Pentru un sir cu elemente sortate crescator aproape complet\*:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nr de elemente | T(n) în secunde | | | |
|  | Test1 | Test2 | Test3 | Media |
| 10 | 0.0 | 0.0 | 0.0 | 0.0 |
| 100 | 0.0 | 0.0 | 0.0 | 0.0 |
| 1000 | 0.008005 | 0.008977 | 0.008976 | 0.005984 |
| 10\_000 | 0.099761 | 0.108856 | 0.103700 | 0.070852 |
| 100\_000 | 1.207779 | 1.218782 | 1.173338 | 1.199966 |
| 1\_000\_000 | 14.310509 | 14.458778 | 14.194284 | 14.321190 |
| 100\_000\_000 | 1812.894613 | - | - | 1812.894613 |

\* n-n/4 elemente sortate și restul nesortate

Pentru un sir cu elementele sortate descrescator:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nr de elemente | T(n) în secunde | | | |
|  | Test1 | Test2 | Test3 | Media |
| 10 | 0.0 | 0.0 | 0.0 | 0.0 |
| 100 | 0.0 | 0.0 | 0.0 | 0.0 |
| 1000 | 0.007977 | 0.008947 | 0.008006 | 0.016953 |
| 10\_000 | 0.104631 | 0.096741 | 0.101728 | 0.101033 |
| 100\_000 | 1.243573 | 1.216839 | 1.200453 | 1.220288 |
| 1\_000\_000 | 13.565044 | 13.778510 | 13.775048 | 13.706200 |
| 100\_000\_000 |  | - | - |  |

**Concluzie:** Algoritmul este unul foarte eficient, reușind să se descurce atât cu seturi mici de date cât și cu seturi mari de date. Dezavantajul însă este că ocupă mult spațiu în memorie comarativ cu algoritmii anteriori. Un lucru interesant ce se poate observa analizând tabele este că pentru seturi de peste 106 algoritmul pare mai eficient în momentul în care este sortat descrescător, iar pentru seturi mai mici de 106 algoritmul este mai eficient când este aproape sortat.

**Sortarea „Quick sort”**

Timpul de executie: *T(n) = nlogn*

Ordinul de complexitate: *O(nlogn)*

Spațiu de memorie: *O(nlogn)*

**Date experimentale:**

Pentru un sir cu elemente nesortate:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nr de elemente | T(n) în secunde | | | |
|  | Test1 | Test2 | Test3 | Media |
| 10 | 0.0 | 0.0 | 0.0 | 0.0 |
| 100 | 0.0 | 0.0 | 0.0 | 0.0 |
| 1000 | 0.005979 | 0.005984 | 0.004986 | 0.01097 |
| 10\_000 | 0.087728 | 0.069784 | 0.070810 | 0.076107 |
| 100\_000 | 0.908356 | 0.853559 | 0.914571 | 0.892162 |
| 1\_000\_000 | 11.101345 | 11.295309 | 11.164727 | 11.187127 |
| 100\_000\_000 | >60min | - | - | >60min |

Pentru un sir cu elemente sortate crescator aproape complet\*:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nr de elemente | T(n) în secunde | | | |
|  | Test1 | Test2 | Test3 | Media |
| 10 | 0.0 | 0.0 | 0.0 | 0.0 |
| 100 | 0.001993 | 0.001996 | 0.001992 | 0.001993 |
| 1000 | 0.139134 | 0.137660 | 0.147633 | 0.141475 |
| 10\_000 | - | - | - | - |
| 100\_000 | - | - | - | - |
| 1\_000\_000 | - | - | - | - |
| 100\_000\_000 | - | - | - | - |

\* n-n/4 elemente sortate și restul nesortate

Pentru un sir cu elementele sortate descrescator:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nr de elemente | T(n) în secunde | | | |
|  | Test1 | Test2 | Test3 | Media |
| 10 | 0.0 | 0.0 | 0.0 | 0.0 |
| 100 | 0.001996 | 0.002021 | 0.001996 | 0.002004 |
| 1000 | - | - | - | - |
| 10\_000 | - | - | - | - |
| 100\_000 | - | - | - | - |
| 1\_000\_000 | - | - | - | - |
| 100\_000\_000 | - | - | - | - |

**Concluzie:** Din punct de vedere al timpului de execuție algoritmul este la fel de eficient ca Sortarea prin interclasare, dar este mai eficient datorită spațiului mic de care are nevoie (sortearea prin inteclasare are nevoie de O(n) spațiu de memorie, iar sortarea rapidă are nevoie de O(1) ). Un minus pentru această sortare este numărul de recursiuni necesar. Acest minus m-a împiedicat să calculez input-uri mai mari de 10.000 în momentul în care șirul este sortat parțial, respectiv 1000 când șirul este sortat descrescător.

**Sortarea prin numărare**

Acest algoritm nu se bazează pe compararea dintre elemente, ci presupune că elementele se află într-un interval anume și creează o listă auxiliară pe baza căreia să rezulte lista sortată.

Ordinul de complexitate: *O(n+k)*

Spațiu de memorie: *O(1)*

**Date experimentale:**

Pentru un sir cu elemente nesortate:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nr de elemente | T(n) în secunde | | | |
|  | Test1 | Test2 | Test3 | Media |
| 10 | 0.0 | 0.0 | 0.0 | 0.0 |
| 100 | 0.000962 | 0.000998 | 0.001024 | 0.00994 |
| 1000 | 0.001994 | 0.001979 | 0.000996 | 0.001656 |
| 10\_000 | 0.019944 | 0.020976 | 0.020948 | 0.020622 |
| 100\_000 | 0.202266 | 0.203428 | 0.200520 | 0.202071 |
| 1\_000\_000 | 2.156402 | 2.143473 | 2.190582 | 2.163485 |
| 100\_000\_000 | 197.029090 | - | - | 197.029090 |

Pentru un sir cu elemente sortate crescator aproape complet\*:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nr de elemente | T(n) în secunde | | | |
|  | Test1 | Test2 | Test3 | Media |
| 10 | 0.0 | 0.0 | 0.0 | 0.0 |
| 100 | 0.0 | 0.0 | 0.0 | 0.0 |
| 1000 | 0.001994 | 0.000997 | 0.001965 | 0.001652 |
| 10\_000 | 0.019946 | 0.017952 | 0.019950 | 0.019282 |
| 100\_000 | 0.192482 | 0.186534 | 0.191519 | 0.190178 |
| 1\_000\_000 | 2.013075 | 1.940320 | 1.948983 | 1.967459 |
| 100\_000\_000 | 169.519162 | - | - | 169.519162 |

\* n-n/4 elemente sortate și restul nesortate

Pentru un sir cu elementele sortate descrescator:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nr de elemente | T(n) în secunde | | | |
|  | Test1 | Test2 | Test3 | Media |
| 10 | 0.0 | 0.0 | 0.0 | 0.0 |
| 100 | 0.0 | 0.0 | 0.0 | 0.0 |
| 1000 | 0.001992 | 0.001993 | 0.001994 | 0.001993 |
| 10\_000 | 0.019916 | 0.017980 | 0.019945 | 0.019280 |
| 100\_000 | 0.192485 | 0.185609 | 0.193032 | 0.126213 |
| 1\_000\_000 | 1.897666 | 1.881885 | 1.864995 | 1.881515 |
| 100\_000\_000 | 164.952610 | - | - | 164.952610 |

**Concluzie:** Pe baza testelor rulate de mine am ajuns la concluzia că algoritmul este foarte eficient fiind linear si având un ordin de complexitate O(n+k) unde k reprezintă marginea superioară a intervalului între care se află numerele din șir. Pe baza testelor pe care le-am făcut eu algoritmul este foarte eficent, dar își pierde din eficiență în momentul în care k este un numar foarte mare, spre exemplu n2. Atunci algoritmul este de complexitate O(n2). Faptul că algoritmul nu se folosește de comparții ci de o anumită codificare nu funcționează la fel pe toate seturile de date. Spre exemplu, seturile folosite de mine au fost formate doar din numere pozitive, iar algoritmul a fost adaptat pentru numere pozitive (folosind un dicționar care să rețină valorile). Însă dacă dorim să folosim și numere negative, atunci algoritmul trebuie adaptat. Lucru care nu este necesar când folosim un algoritm bazat pe comparații.

**Soratarea „Cocktail sort”**

Algoritmul reprezintă o variație a algoritmului „Bubble sort” exceptând faptul că traversează sirul în abele direcții alternativ. Parcurge șirul de la stânga la dreapta ducând cel mai mare element pe ultima poziție. Când ajunge în capăt se întoarce aducând cel mai mic element pe prima poziție. Procesul se repetă până când șirul este sortat complet.

Ordinul de complexitate: *O(n2)*

Spațiu de memorie: *O(1)*

**Date experimentale:**

Pentru un sir cu elemente nesortate:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nr de elemente | T(n) în secunde | | | |
|  | Test1 | Test2 | Test3 | Media |
| 10 | 0.0 | 0.0 | 0.0 | 0.0 |
| 100 | 0.002990 | 0.002962 | 0.002991 | 0.002981 |
| 1000 | 0.237364 | 0.246341 | 0.223433 | 0.235712 |
| 10\_000 | 23.792251 | 24.197489 | 24.006752 | 23.998830 |
| 100\_000 | 2535.8485 | - | - | 2535.8485 |
| 1\_000\_000 | >40min | - | - | >40min |
| 100\_000\_000 | >40min | - | - | >40min |

Pentru un sir cu elemente sortate crescator aproape complet\*:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nr de elemente | T(n) în secunde | | | |
|  | Test1 | Test2 | Test3 | Media |
| 10 | 0.0 | 0.0 | 0.0 | 0.0 |
| 100 | 0.0 | 0.0 | 0.0 | 0.0 |
| 1000 | 0.044878 | 0.042913 | 0.044873 | 0.044221 |
| 10\_000 | 4.538125 | 4.570798 | 4.655349 | 4.588090 |
| 100\_000 | 452.810191 | - | - | 452.810191 |
| 1\_000\_000 | >40min | - | - | >40min |
| 100\_000\_000 | >40min | - | - | >40min |

\* n-n/4 elemente sortate și restul nesortate

Pentru un sir cu elementele sortate descrescator:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nr de elemente | T(n) în secunde | | | |
|  | Test1 | Test2 | Test3 | Media |
| 10 | 0.0 | 0.0 | 0.00 | 0.0 |
| 100 | 0.003991 | 0.002992 | 0.003989 | 0.003657 |
| 1000 | 0.402952 | 0.407914 | 0.400928 | 0.403931 |
| 10\_000 | 39.561781 | 39.551581 | 40.343138 | 39.818833 |
| 100\_000 | 3810.995640 | - | - | 3810.995640 |
| 1\_000\_000 | >40min | - | - | >40min |
| 100\_000\_000 | >40min | - | - | >40min |

Ordinul de complexitate în cel mai favorabil caz (când elementele sunt sortate crescător) este O(n), iar în cel mai nefavorabil caz este O(n2) (când elementele sunt sortate descrescător).

**Concluzie:** Dacă comparăm timpii de execuție putem observa că algoritmul este mai ineficient pentru seturi de date mai mici decât „Bubble sort”, dar câștigă în eficiență când vine vorba de seturi de date mai mari.

**Sortarea „Tim sort”**

Algoritmul se bazează pe sortarea prin inserție și sortarea prin interclasare astfel:

* Se împarte șirul de date în date subșiruri
* Fiecare subșir este apoi sortat folosind sortarea prin inserție
* Toate șirurile sunt mai apoi „unite” folosind sortarea prin interclasare

Ordinul de complexitate: *O(nlogn)*

Spațiu de memorie: *O(n)*

De menționat un fapt interesant: acest algoritm este folosit de limbajul de programare python în funcțiile *sort* și *sorted*.

**Date experimentale:**

Pentru un sir cu elemente nesortate:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nr de elemente | T(n) în secunde | | | |
|  | Test1 | Test2 | Test3 | Media |
| 10 | 0.0 | 0.0 | 0.0 | 0.0 |
| 100 | 0.000997 | 0.000994 | 0.000997 | 0.000996 |
| 1000 | 0.009974 | 0.009945 | 0.010001 | 0.009973 |
| 10\_000 | 0.120679 | 0.123698 | 0.119708 | 0.121361 |
| 100\_000 | 1.599111 | 1.565726 | 1.543192 | 1.569343 |
| 1\_000\_000 | 19.747964 | 20.162880 | 19.589665 | 19.833503 |
| 100\_000\_000 | 2722.173400 | - | - | 2722.173400 |

Pentru un sir cu elemente sortate crescator aproape complet\*:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nr de elemente | T(n) în secunde | | | |
|  | Test1 | Test2 | Test3 | Media |
| 10 | 0.0 | 0.0 | 0.0 | 0.0 |
| 100 | 0.0 | 0.0 | 0.0 | 0.0 |
| 1000 | 0.006011 | 0.005014 | 0.004986 | 0.005337 |
| 10\_000 | 0.086735 | 0.082781 | 0.090786 | 0.086767 |
| 100\_000 | 1.198425 | 1.074841 | 1.104177 | 1.125814 |
| 1\_000\_000 | 14.159169 | 14.061060 | 13.995148 | 14.071792 |
| 100\_000\_000 | 2051.790194 | - | - | 2051.790194 |

\* n-n/4 elemente sortate și restul nesortate

Pentru un sir cu elementele sortate descrescator:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nr de elemente | T(n) în secunde | | | |
|  | Test1 | Test2 | Test3 | Media |
| 10 | 0.0 | 0.00 | 0.0 | 0.0 |
| 100 | 0.001000 | 0.000997 | 0.001025 | 0.001007 |
| 1000 | 0.014958 | 0.015985 | 0.014959 | 0.015300 |
| 10\_000 | 0.149150 | 0.140623 | 0.138657 | 0.14281 |
| 100\_000 | 1.838622 | 1.893590 | 1.841146 | 1.857786 |
| 1\_000\_000 | 22.858464 | 22.553965 | 22.760815 | 22.724414 |
| 100\_000\_000 | 2599.132677 | - | - | 2599.132677 |

Ordinul de complexitate în cel mai favorabil caz (când elementele sunt sortate crescător) este O(n), iar în cel mai nefavorabil caz este O(nlogn) (când elementele sunt sortate descrescător).

**Concluzie:** Ideea de bază al acestui algoritm de sortare este că sortarea prin inserție se descurcă bine pe seturi de date mici, astfel ajungându-se la timpi de execuție mai buni. Diferențele între acest algoritm și sortarea prin interclasare nu sunt mari, dar sunt vizibile pentru seturi de date mai mari de 105, sortarea prin interclasare fiind mai bună.

**Câteva concluzii finale**

1. Counting sort este un algoritm foarte bun, mai ales când avem date de intrare clar specificate și reușim să îl adaptăm corespunzător. Este un algoritm sensibil la imprevizibilitate, motiv pentru care nu este cel mai bun algoritm de sortare overall.
2. Quick sort este un algoritm de sortare foarte bun, având un timp de execuție mic și ocupând puțin spațiu de memorie, dar din cauza numărului mare de recursiuni nu reprezintă cel mai bun algoritm de sortare.
3. Merge sort reprezintă cel mai bun algoritm de sortare, în special când lucrezi cu date mari. Minusul acestui algoritm constă în faptul că ocupă mult spațiu de memorie, iar dacă lucrezi și cu date mari și cu multe date...
4. Cel mai bun algoritm de sortare când lucrezi cu date aproape sortate sau cu date puține este sortarea prin inserție.
5. Dacă vrei să lucrezi pe un set mic de date, sortarea „Bubble Sort” sau sortarea prin selecție sunt doi algoritmi simpli cu spațiu de memorie mic, astfel sunt numai buni.
6. Diferența dintre „Bubble sort” și „Cocktail sort” este că prima este mai bună pentru seturi de date mici, iar a doua este mai bună pentru seturi de date mai mari.