Algoritmi divide et impera

Tehnica divide et impera

Divide et impera este o tehnică de elaborarea a algoritmilor ce constă în:

- Descompunerea problemei ce trebuie rezolvată într-o serie de subprobleme mai mici.
- Rezolvarea succesivă și independentă a fiecăreia din aceste probleme și găsirea unei serii de subsoluții.
- Recompunerea subsoluțiilor pentru a găsi soluția cazului inițial.

Să presupunem că avem un algoritm *A* cu timp pătratic. Fie *c* o constantă, astfel încât timpul pentru a rezolva un caz de mărime n este $t_A(n) \leq cn^2$. Să presupunem că este posibil să rezolvăm un astfel de caz prin descompunerea în trei subcazuri, fiecare de mărime $\lceil n/2 \rceil$. Fie d o constantă, astfel încât timpul necesar pentru descompunere și recompunere să fie $t(n) \leq dn$. Folosind vechiul algoritm și ideea de descompunere-recompunere a subcazurilor, obținem un nou algoritm *B* pentru care:

$$t_B(n) = 3t_A\left(\left\lceil \frac{n}{2} \right\rceil\right) + t(n) \leq 3c\left(\frac{n+1}{2}\right)^2 + dn = \frac{3}{4}cn^2 + \left(\frac{3}{2} + d\right)n + \frac{3}{4}c$$

Termenul $\frac{3}{4}cn^2$ domină ceilalți termeni atunci când n este suficient de mare, ceea ce înseamnă că algoritmul B este cel puțin cu 25% mai rapid decât algoritmul *A*. Totuși ordinul de timp al lui *B* este pătratic.

Putem continua în mod recursiv acest procedeu, împărțind subcazurile în subsubcazuri ș.a.m.d. Pentru subcazurile care nu sunt mai mari decât un prag n_0, vom folosi tot algoritmul *A*. Obținem astfel un alt algoritm *C* cu ordinul de timp:

$$t_C(n) = \begin{cases} t_A(n) & pentru\ n \leq n_0 \\ 3t_C\left(\left\lceil \frac{n}{2} \right\rceil\right) & pentru\ n > n_0 \end{cases}$$

Se poate calcula că $t_C(n)$ este în ordinul de timp n^{lg3}, adică un timp mai bun decât cel pătratic.

Mai jos avem o descriere generală a problemei divide et impera:

$FUNCTION\ divimp(x)$

1. $\triangleright\ returnează\ o\ soluţie\ pentru\ cazul\ x$
2. $\boldsymbol{if}\ x\ este\ suficient\ de\ mic\ \boldsymbol{then\ return}\ adhoc(x)$
3. $\triangleright\ descompune\ x\ în\ subcazurile\ x_1, x_2, \dots, x_k$
4. $\boldsymbol{for}\ i \leftarrow 1\ \boldsymbol{to}\ k\ \boldsymbol{do}\ y_i \leftarrow divimp(x_i)$
5. $\triangleright\ recompune\ y_1, y_2, \dots, y_k\ în\ scopul\ obţinerii\ soluţiei\ pentru\ x$
6. $\boldsymbol{return}\ y$

Unde $adhoc$ este subalgoritmul de bază folosit pentru rezolvarea micilor subcazuri ale problemei în cauză (în exemplul nostru, acest subalgoritm a fost *A*) .

Un algoritm de divide et impera trebuie să evite descompunerea recursivă a subcazurilor "suficient de mici", deoarece, pentru acestea, este mai eficientă aplicarea directă a subalgoritmului de bază. Ce înseamnă însă "suficient de mic"?

În exemplul precedent, cu toate că valoarea lui n_0 nu influențează ordinul de timp, este influențată însă constanta multiplicativă a lui n^{lg3}, ceea ce poate acea un rol considerabil în eficiența algoritmului. Pentru un algoritm divide et impera oarecare, chiar dacă ordinul de timp nu poate fi îmbunătățit, se dorește optimizarea acestui prag în sensul obținerii unui algoritm cât mai eficient. Nu există o metodă teoretică generală pentru aceasta, pragul optim depinzând nu numai de algoritmul în cauză, dar și de particularitatea implementării. Considerând o implementare dată, pragul optim poate fi determinat empiric, prin măsurarea timpului de execuție pentru diferite valori ale lui n_0 și cazuri de mărimi diferite.

În general, se recomandă o metodă hibridă, care constă în

- determinarea teoretică a formei ecuațiilor recurente
- găsirea empirică a valorilor constantelor folosite de aceste ecuații

Revenind la exemplul inițial, pragul optim poate fi găsit rezolvând ecuația

$$t_A(n) = 3t_A\left(\left\lceil\frac{n}{2}\right\rceil\right) + t(n)$$

Empiric se găsește $n_0 \cong 67$, adică valoarea pentru care nu mai are importanță dacă aplicăm algoritmul *A* direct, sau continuăm descompunerea. Observăm că metoda este prin definiție recursivă. Uneori este posibil să eliminăm recursivitatea printr-un ciclu iterativ. Este indicat, ca atunci când se poate, recursivitatea să fie înlocuită cu metode iterative, deoarece se economisește memorie, și se poate evita problema de stack overflow ce poate apare în cazul recursivității.

Quicksort

Quicksort este un algoritm de sortarea a cărui ordin de timp, în cazul cel mai nefavorabil este de $\theta(n^2)$ pentru un șir de n numere. În ciuda acestui fapt, acest algoritm este folosi ca *best practice* pentru sortarea unei structuri de date, deoarece este eficient în cazul mediu. Timpul său mediu este $\theta(n \log n)$. De asemenea constantele din ordinul de timp sunt relativ mici.

Alt avantaj este faptul că folosește sortarea *in place*. Acest lucru înseamnă că sortarea are loc direct pe elementele șirului dat, fără folosirea unui alt șir auxiliar. Avantajul este că memoria virtuală este economisită.

Descrierea algoritmului

Să presupunem că avem un subșir de $r - p + 1$ elemente $A[p..r]$ ce trebuie sortat. Iată rezolvarea propusă de acest algoritm, în conformitate cu metoda divide et impera:

Divide: șirul $A[p..q]$ este partiționat (rearanjat) în două subșirului nevide, $A[p..q]$ și $A[q+1..r]$, astfel ca fiecare element din $A[p..q]$ să fie mai mic sau egal ca oricare element din $A[q+1..r]$. Indexul q este calculat conform procedurii de partiționare de mai jos.

Cucerește: cele două subșiruri $A[p..q]$ și $A[q+1..r]$ sunt sortate prin apeluri recursive ale aceleiași proceduri de quicksort.

Combină: din moment ce subșirurile sunt sortate în același șir, nu mai trebuie să le combinăm, șirul $A[p..r]$ este sortat și reprezintă soluția.

Mai jos avem procedura de implementare a algoritmului quicksort

$QUICKSORT(A, p, r)$

1. $\boldsymbol{if}\ p < r$
2. $\quad then\ q \leftarrow PARTITION(A, p, r)$
3. $\quad QUICKSORT(A, p, q)$
4. $\quad QUICKSORT(A, q+1, r)$

Pentru a sorta întregul șir A, apelul inițial se va face cu parametrii $QUICKSORT(A, 1, length[A])$.

Partiționarea șirului

Cheia întregului algoritm constă în această procedură care rearanjează șirul $A[p..r]$ după cum urmează.

$PARTITION(A, p, r)$

1. $x \leftarrow A[p]$
2. $i \leftarrow p - 1$
3. $j \leftarrow r + 1$
4. $\boldsymbol{while}\ TRUE\ \boldsymbol{do}$
5. $\quad\boldsymbol{repeat}\ j \leftarrow j - 1$
6. $\quad\quad\boldsymbol{until}\ A[j] \leq x$
7. $\quad\boldsymbol{repeat}\ i \leftarrow i + 1$
8. $\quad\quad\boldsymbol{until}\ A[i] \geq x$
9. $\quad\boldsymbol{if}\ i < j$
10. $\quad\quad\boldsymbol{then}\ interschimbă\ A[i]\ cu\ A[j]$
11. $\quad\quad\boldsymbol{else\ return}\ j$

În figura de mai jos este prezentat modul de funcționare al acestei proceduri.

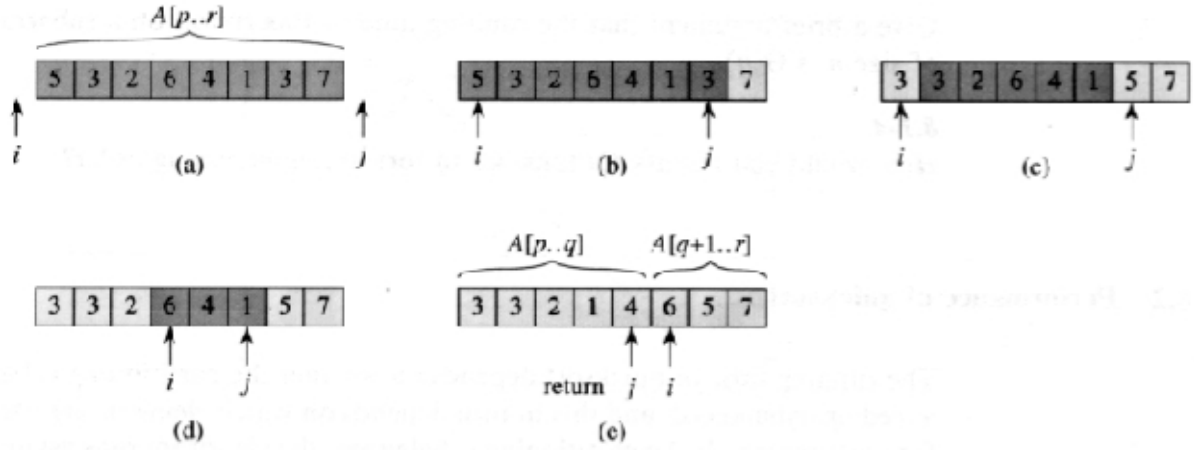

Figura 1. Modul de funcționare al procedurii $PARTITION$ pe un șir simplu. Elementele deschise la culoare au fost plasate în locațiile corecte, iar elementele închise la culoare nu sunt încă la locul final. a) șirul inițial, și valorile lui *i* și *j* la capetele exterioare ale șirului. Pivotul va fi $x = A[p] = 5$. b) pozițiile lui *i* și *j* la linia9 din prima iterație a buclei while. c) rezultatul interschimbării între elementele de pe pozițiile *i* și *j* corespunzătoare liniei 10 din algoritm. d) pozițiile lui *i* și *j* la linia 9, după ce a fost executată odată, bucla while. e) pozițiile lui *i* și *j* după a treia execuție a buclei while. procedura se încheie deoarece $i \geq j$, și se returnează valoarea $q = j$. În clipa aceasta elementele de la stânga lui *j*, inclusiv, sunt mai mici sau egale cu $x = 5$, iar cele de la dreapta sunt mai mari sau egale cu $x = 5$.

Algoritmul va forța în repetiții succesive obținerea a două subșiruri cu elementele celui din stânga mai mici decât cele aparținând celui din dreapta. Procedura *PARTITION* determină acest lucru astfel:

Prima dată selectează un element $x = A[p]$ din șirul $A[p..r]$ ca element *pivot* pentru a rearanja acest șir. Apoi va dezvolta două regiuni $A[p..i]$ și $A[j..r]$ de la începutul șirului, respectiv de la sfârșitul șirului, astfel ca fiecare element din $A[p..i]$ sa fie mai mic sau egal cu orice element din $A[j..r]$, sau x care este reprezentantul lui $A[j..r]$, deoarece fiecare element din acest subșir este mai mare sau egal cu x. Inițial $i = p - 1$ și $j = r + 1$, astfel că cele două subșiruri sunt inițial vide.

În cadrul corpului lui *while*, indexul *j* este decrementat și indexul este incrementat, până când $A[i] \geq x \geq A[j]$. Presupunând că aceste inegalități sunt stricte, $A[i]$ este prea mare ca el să se afle în partea de jos a șirului, iar $A[i]$ este prea mic ca el să se afle în partea superioară șirului. De aceea, se efectuează interschimbarea între $A[i]$ și $A[j]$ ce va reface ordinea firească.

Instrucțiunile lui while se repetă până când $i \geq j$, moment în care sigur șirul $A[p..r]$ a fost partiționat în două subșiruri $A[p..q]$ și $A[q+1..r]$ cu $p \leq q < r$ astfel ca nici un element din $A[p..q]$ să nu fie mai mare decât oricare element din $A[q+1..r]$.

Ordinul de timp al aceste proceduri este $\theta(n)$ unde $n = r - p + 1$.

Performanta algoritmului quicksort

Timpul de rulare al acestui algoritm depinde de partiționare: dacă este sau nu realizată într-un mod balansat. Aceasta înseamnă că elementele din stânga sunt aproximativ egale cu cele din dreapta. Dacă partiționarea este una balansată, algoritmul are timp de rulare $\theta(n) = n \log n$, în caz contrar ordinul de timp fiind unul pătratic.

Cel mai nefavorabil caz

Acesta are loc atunci când, procedura de partiționare produce o regiune cu $n - 1$ elemente și alta cu 1 element. Să presupunem că acest mod de partiționare va apare la fiecare pas din algoritm (în fiecare apel recursiv al procedurii $QUICKSORT$). Din moment ce partiționare va avea un timp liniar $\theta(n)$ și $T(1) = \theta(1)$, formula de recurență pentru timpul total este:

$$T(n) = T(n-1) + \theta(n)$$

Pentru a evalua această recurență utilizăm iterațiile:

$$T(n) = T(n-1) + \theta(n)$$

$$T(n) = T(n-2) + \theta(n-1) + \theta(n)$$

...

$$T(n) = \sum_{k=1}^{n} \theta(k)$$

$$T(n) = \theta \sum_{k=1}^{n} k$$

$$(n) = \theta(n^2)$$

În Figura 2, este prezentat al mod de a calcula timpul quicksort în cazul cel mai nefavorabil

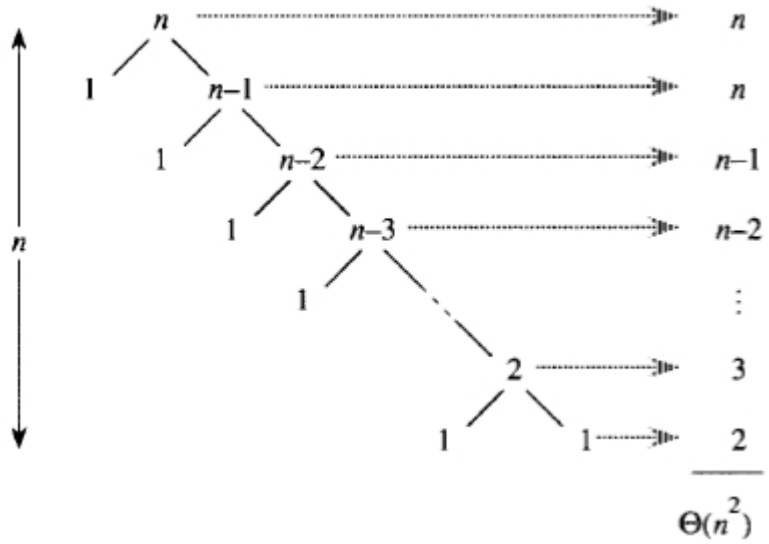

Figura 2. Arborele de recursivitate pentru *QUICKSORT* pentru care *PARTITION* separă șirul în 1 respectiv n-1 elemente. Timpul algoritmului este în $\theta(n^2)$.

Cel mai bun caz

Dacă procedura de partiționare ar produce două subșiruri de mărime $n/2$, atunci algoritmul ar merge mult mai repede. Recurența este dată de $T(n) = 2T\left(\frac{n}{2}\right) + \theta(n)$ ce are ca soluție $T(n) = n \log n$. În Figura 3 este prezentat arborele de recursivitate în acest caz.

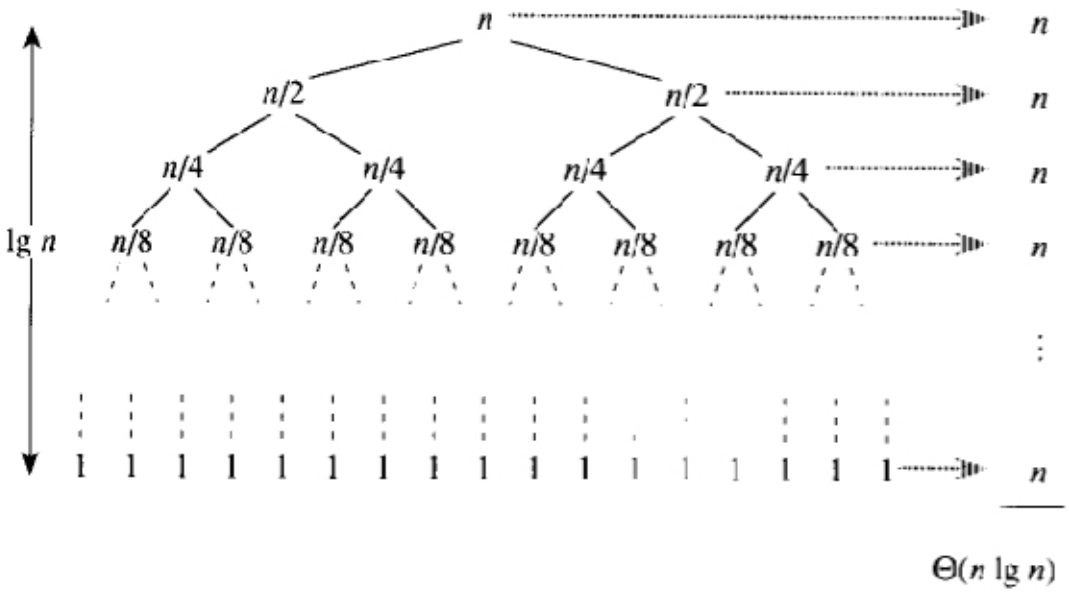

Figura 3. Arborele de recurență pentru *QUICKSORT* atunci când *PARTITION* produce două subșiruri de mărimi egale. Timpul algoritmului este $\theta(n \log n)$.

Cazul mediu

Să luăm acum cazul cel mai probabil, adică cel în care este posibil ca elementele să nu fie nici perfect balansate și nici total nebalansate. Să presupunem că, de exemplu, *PARTITION* produce o repartizare de 90% elemente într-o parte, 10% în cealaltă, dintr-un total de 100% elemente.

Vom obține o recurență $T(n) = T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + n$, unde am înlocuit $\theta(n)$ cu n.

Fiecare nivel de recurență are un cost total de n, până când condiția limită este îndeplinită pentru o adâncime de $\log_{10} n = \theta(\log n)$. Astfel costul total este $\theta(n \log n)$. Deși împărțirea de 9 la 1 părea complet nebalansată, se poate constata că algoritmul este tot în timp $n \log n$, doar că baza diferă, însă aceasta nu este semnificativă.

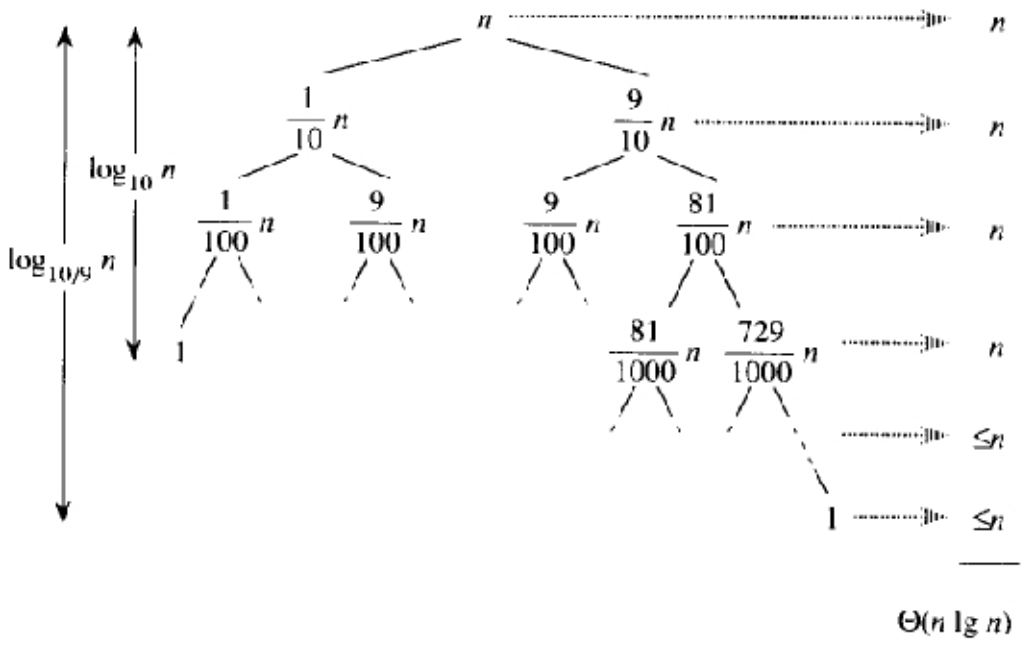

Figura 4. Arborele de recursivitate pentru QUICKSORT în care PARTITION produce întotdeauna o repartizare de 9 la 1. Timpul algoritmului este $\theta(n \log n)$.

Algoritmul Mergesort

Fie $T[1..n]$ un tablou pe care dorim să îl sortăm crescător. Prin tehnica divide et impera putem proceda astfel: separăm tabloul T în două părți de mărimi cât mai apropiate, sortăm aceste părți prin apeluri recursive, apoi interclasăm soluțiile pentru fiecare parte, astfel ca elementele să fie ordonate crescător. Soluția problemei poate fi formulată astfel:

Divide: Împarte secvența de n elemente în două subsecvențe de $\frac{n}{2}$ elemente.

Cucerește: Sortează două subsecvențe recursiv folosind mergesort

Combină: îmbină cele două subsecvențe pentru a rezolva problema.

Procedura de sortare este aceasta:

$MERGE - SORT(A, p, r)$

1. $\boldsymbol{if}\ p < r\ \boldsymbol{then}$
2. $\quad q \leftarrow \left\lfloor \frac{p+r}{2} \right\rfloor$
3. $\quad MERGE - SORT(A, p, q)$
4. $\quad MERGE - SORT(A, q+1, r)$
5. $\quad MERGE(A, p, q, r)$

Pentru a sorta întregul șir A, trebuie să apelăm $MERGE - SORT(A, 1, length[A])$.

Procedura $MERGE$ combină soluțiile și anume cele două subșiruri care sunt gata sortate după ce se va fi apelat $MERGE - SORT$ de la pasul 3 respectiv 4. De fapt se va face o interclasare a celor două subșiruri rezultând un șir $A[p..r]$ ce conține elementele din $A[p..q]$ și $A[q+1..r]$ sortate printr-un procedeu cunoscut de sortare (de exemplu interclasare).

Dacă ne uităm la operațiile din această procedură, când n este o putere de doi, algoritmul constă în combinarea unor perechi de câte un element astfel încât se formează secvențe de lungime 2, iar apoi se combină secvențe de câte 2 elemente în secvențe de 4 elemente și așa mai departe până când ajungem la secvența finală de n elemente. Mai jos avem o exemplificare a rulării acestui algoritm:

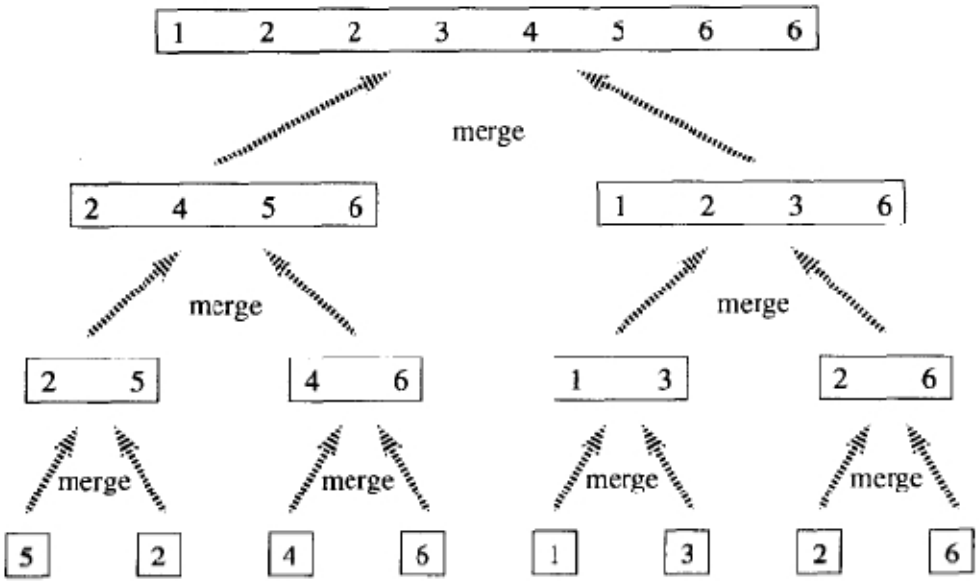

Figura 5. Operația de mergesort pentru un șir $A = \langle 5,2,4,6,1,3,2,6 \rangle$. Lungimea secvenței sortate crește pe măsură ce algoritmul avansează: adică revine din apelurile recursive către apelul inițial.

Analiza recurentei acestui algoritm

Fie $T(n)$ ordinul de timp pentru o problemă cu un șir de mărime n. Dacă mărimea problemei este mică vom spune că $n \leq c$ pentru o constantă c, și atunci ordinul de timp este constant în $\theta(1)$. Să presupunem că divizăm problema în a subprobleme, atunci file ca fiecare subproblemă să aibă ca intrare un șir de mărime $\frac{1}{b}$ din original. Fie $D(n)$ timpul necesar împărțirii problemei în subprobleme și $C(n)$ timpul necesar combinării acestora obținem recurența:

$$T(n) = \begin{cases} \theta(1) & dacă\ n \leq c \\ aT\left(\frac{n}{b}\right) + D(n) + C(n) & altfel \end{cases}$$

Aceasta este formula general valabilă. Să considerăm acum cazul particular pentru mergesort:

Divide: pasul de împărțire a șirului în două, este constant deci $D(n) = \theta(1)$

Cucerește: vom rezolva recursiv două subprobleme, fiecare de mărime $n/2$ ce vor contribui cu $2T(n/2)$ la timpul total.

Combină: Procedura $MERGE$ va necesita un timp liniar deci $\theta(n)$.

Atunci când adăugăm funcțiile de timp $D(n)$ și $C(n)$ formulei de recurență de mai sus, obținem această formulă:

$$T(n) = \begin{cases} \theta(1) & dacă\ n = 1 \\ 2T\left(\frac{n}{2}\right) + \theta(n) & dacă\ n > 1 \end{cases}$$

Vom rezolva această recurență prin iterații repetate astfel:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

$$T(n) = 2\left(2T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n = 4T\left(\frac{n}{4}\right) + 2n$$

$$T(n) = 2^2\left(2T\left(\frac{n}{8}\right) + \frac{n}{4}\right) + 2n = 8T\left(\frac{n}{8}\right) + 3n$$

Observăm deja un șablon de formare a acestei recurențe astfel că

$$T(n) = 2^{\log n}T\left(\frac{n}{n}\right) + n\log n = n + n\log n$$

Astfel ordinul de timp al acestui algoritm este $\theta(n\log n)$.

Radix sort

Radix sort este algoritmul folosit inițial de computerele ce rulau programe aflate pe cartele, pentru a le sorta. Cartelele erau organizate în 80 de coloane și în fiecare coloană o gaură putea fi făcută într-unul din cele 12 locuri disponibile. Aparatul care sorta aceste carduri, putea fi programat să inspecteze pe o coloană, una din cele 12 găuri pentru a găsi locația. Un operator avea să culeagă cardurile, astfel ca primele să aibă gaura în locația de sus a cardului, apoi cele ce urmau să aibă gaura în locația a doua de sus, și așa mai departe.

Pentru numere în zecimal, se vor folosi 10 locuri pentru fiecare coloană. Un număr cu d digitale ar ocupa un câmp de d coloane. Din moment ce un aparat de sortare nu poate analiza decât o coloană la un moment dat, problema sortării a n carduri de un număr pe d digitale se poate rezolva astfel.

Intuitiv, se pot sorta numerele după cel mai semnificant bit al lor, separa pe coloane, și apoi combina rezultatul recursiv pentru a obține ordinea finală.

Radixsort rezolvă această problemă prin sortarea celui mai puțin semnificant bit. Cardurile sunt combinate astfel: cele cu digitalul 0 precedă pe cele cu digitalui 1 care le precedă pe cele cu digitalui 2 și așa mai departe. Apoi algoritmul trece la următorul pas și anume analizează bitul al doilea după cel mai puțin semnificant bit, aplicându-se aceeași separație ca mai sus.

De remarcat ca la fiecare al k-lea pas al algoritmului, numerele formate cu digitale de până la bitul k sunt deja sortate.

Spre deosebire de celelalte metode această sortare funcționează pe structuri de chei.

Acest algoritm poate funcționa și prin sortarea celui mai semnificant bit, și apoi celui de-al doilea semnificant bit, adică în ordinea inversă decât cea descrisă mai sus, și vom începe cu aceasta.

MSD Radix Sort

Să exemplificăm modul de sortare în cazul în care cheile sunt numere în baza M, de unde provine defapt și numele algoritmului (M=radix). Fie M=2 și unul din numere din șirul de sortat 12.

$$11 = 1 \quad 0 \quad 1 \quad 1$$

Cheile pot fi și alfanumerice, de exemplu șiruri sau diferite caractere.

În figura 6 este exemplificat algoritmul de sortare radix ce ține cont întotdeauna de cel mai din stânga bit, adică MSD(most significant bit).

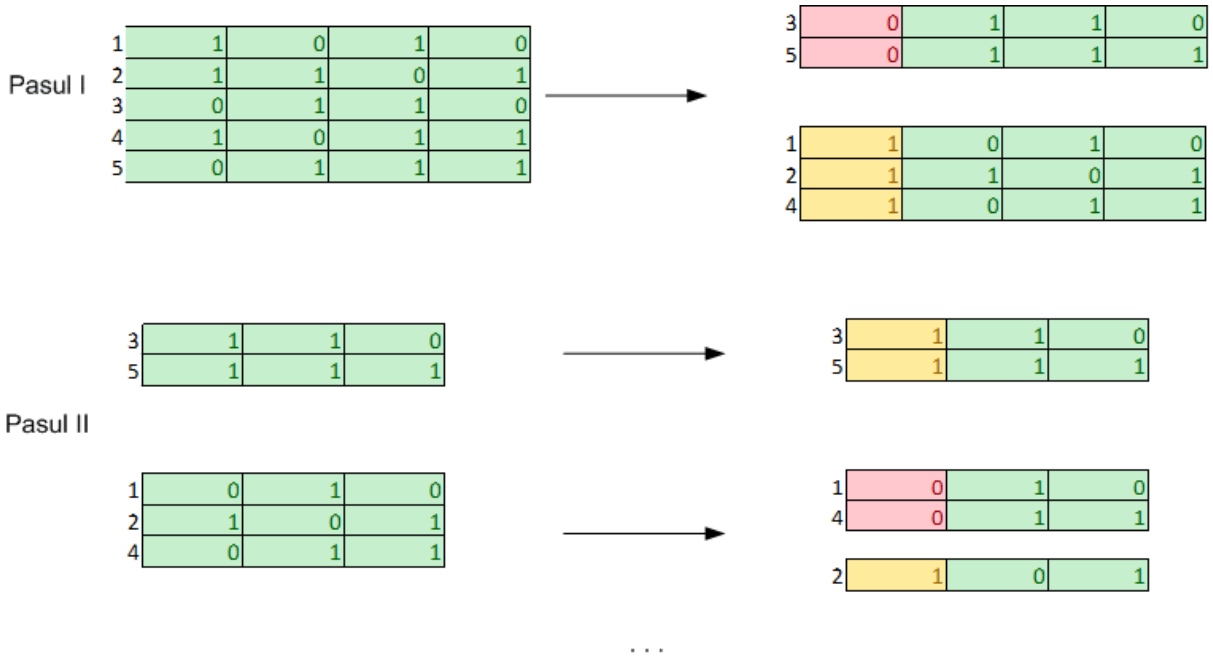

Figura 6. Exemplu pentru sortarea numerelor 10,13,6,11,7. Se transformă în binar fiecare număr în parte, și la primul pas se disociază numerele în două grupe, pe baza bitului de pe prima poziție, adică cele care încep cu 0 într-o grupă, iar cele care încep cu 1 în cealaltă. Se șterge prima coloană, și se reia algoritmul pentru celelalte coloane rămase. Pasul 2 prezintă continuarea algoritmului pentru restul biților din numere și anume începând cu cea de-a doua coloană. Astfel la pasul 2 se elimină din calcul prima coloană deoarece separarea în cele două grupe s-a făcut deja pe baza ei.

Ordinul de timp al acestui algoritm este unde este numărul de biți pe care reprezentăm numerele (va fi dat de cel mai mare număr din șir) iar mărimea șirului de sortat.

Algoritmul este următorul (se bazează pe ideea de partiționare de la quicksort):

1. *repeat*
2. *pe coloană caută de sus în jos un număr ce începe cu 1*
3. *pe coloană caută de jos în sus un număr ce începe cu 0*
4. *interschimbă numerle găsite (dacă există)*
5. *până când indicii de căutare se intersectează*

LSD Radix Sort

Al doilea mod de operare al algoritmului radix sort, este următorul: pornind de la stânga la dreapta, pe coloane vom separa în grupe aparținând aceleiași cifre (în cazul bazei doi în grupe de 0 și 1) după cum urmează. Este important să menținem stabilitatea sortării. Ce semnifică stabilitatea? Sortarea unui șir în mod stabil înseamnă că, pentru două chei egale (să spunem două numere cu aceiași biți dispuși în aceiași ordine), ordinea lor în șir rămâne neschimbată relativ una față de cealaltă. Să luăm următorul exemplu:

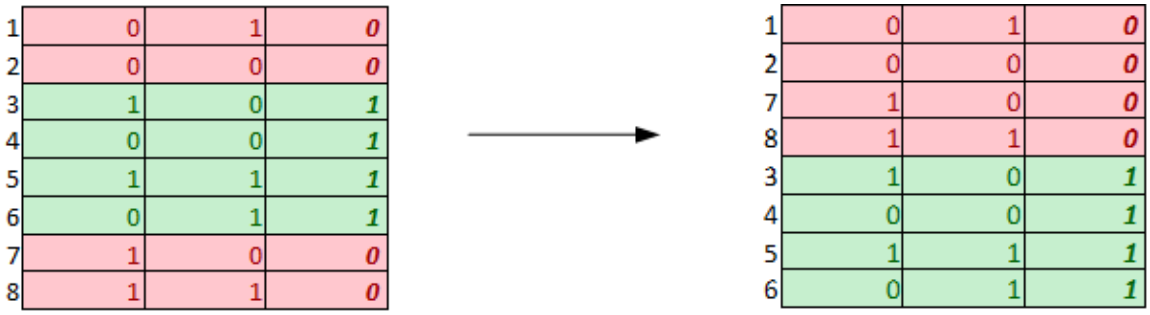

Figura 7. Sortarea radix folosind LSB. Se poate observa că numerele de pe pozițiile 7 și 8, deși au fost mutate conform grupării după cel mai puțin semnificativ bit și anume 0, ele își mențin ordinea inițială relativă din șir.

Mai departe, dacă trecem la al doilea bit după cel mai puțin semnificativ, și avansăm spre stânga de această dată, pentru șirul din figura de mai sus, iată ce obținem:

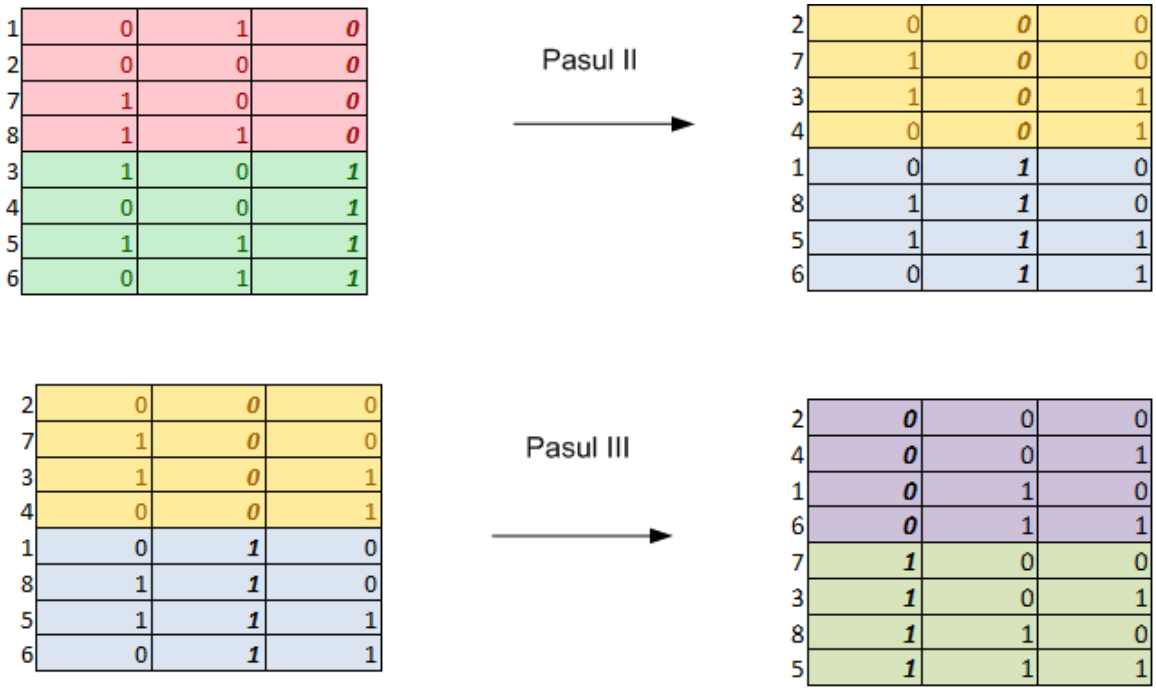

Figura 8. Continuarea algoritmului radix, și obținerea în final a unui șir sortat format din numerele 0-7.

Pentru generalizarea algoritmului, să presupunem că avem un șir de n elemente și anume A, unde cel mai mare dintre numere poate fi reprezentat pe d cifre (în baza *M*), iar digitul 1 este de ordinul cel mai mic iar cel de pe poziția d este de ordinul cel mai mare. Algoritmul este următorul:

$RADIX - SORT(A, d)$

1. $\boldsymbol{for}\ i \leftarrow 1\ \boldsymbol{to}\ d\ \boldsymbol{do}$
2. $sortează\ stabil\ șirul\ A\ după\ digitul\ i$

Corectitudinea acestui algoritm poate fi demonstrată prin inducție, și în principiu se bazează pe faptul ca la al k-lea pas numerele formate din k biți vor fi gata sortate. Analiza ordinului de timp depinde de algoritmul de ordonare stabilă folosit. Dacă nu socotim și algoritmul de sortare, fiecare parcurgere a numerelor de d cifre va lua $\theta(n + k)$ unde k este cifra maximă (în cazul în care $M = 2, k = 1$). Vor fi d pași în total, deci timpul total este $\theta(dn + kd)$. Dacă vom considera d ca fiind constant putem forța notația și afirma că algoritmul are timp liniar.

Totuși, se poate considera că numărul de biți folosiți este în $\theta(\log n)$. Mai concret, să spunem că $d \log n$ este numărul de biți unde $d > 0$ este o constantă. Fiecare număr va fi tratat ca un număr pe d biți. De exemplu, să considerăm sortarea unui milion de numere pe 64 de biți. Dacă reprezentăm numerele pe 4 digitale a câte 2^{16} numere, putem sorta în patru pași folosind sortarea radix. Acest lucru se compară cu un algoritm în $O(n \log n)$. Din păcate, versiunea de sortare radix ce nu folosește sortare in place, va necesita memorie pentru stocarea șirurilor temporare, ceea ce este un dezavantaj față de un quicksort, de exemplu.