LUCRARE DE LABORATOR NR.2

Tema: Metoda divide et impera.

Scopul lucrării:

1. Studierea metodei divide et impera.
2. Analiza şi implementarea algoritmilor bazaţi pe metoda divide et impera.

Note de curs:

1. Tehnica divide et impera

Divide et impera este o tehnica de elaborare a algoritmilor care constă în:

1. Descompunerea cazului ce trebuie rezolvat într-un număr de subcazuri mai mici ale aceleiaşi probleme.

2. Rezolvarea succesivă şi independentă a fiecăruia din aceste subcazuri.

3. Recompunerea subsoluţiilor astfel obţinute pentru a găsi soluţia cazului iniţial.

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 $t_A(n) \le cn^2$. 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 şi recompunere este $t(n) \le dn$. Folosind vechiul algoritm şi ideea de descompunere-recompunere a subcazurilor, obţinem un nou algoritm *B*, pentru care:

$$t_B(n) = 3t_A(\lceil n/2 \rceil)+t(n) \le 3c((n+1)/2)^2+dn = 3/4cn^2+(3/2+d)n+3/4c$$

Termenul $3/4cn^2$ domină pe ceilalţi când *n* este suficient de mare, ceea ce înseamnă ca algoritmul *B* este în esenţă cu 25% mai rapid decât algoritmul *A*. Nu am reuşit însă să schimbăm ordinul timpului, care rămâne pătratic.

Putem să continuăm în mod recursiv acest procedeu, împărţind subcazurile în subsubcazuri etc. Pentru subcazurile care nu sunt mai mari decât un anumit prag n_0, vom folosi tot algoritmul *A*. Obţinem astfel algoritmul *C*, cu timpul

$$t_C(n) = \begin{cases} t_A(n) & \text{pentru } n \le n_0 \\ 3t_C(\lceil n/2 \rceil) + t(n) & \text{pentru } n > n_0 \end{cases}$$

$t_C(n)$ este în ordinul lui $n^{\lg 3}$. Deoarece $\lg 3 \cong 1{,}59$, înseamnă că de această dată am reuşit să îmbunătăţim ordinul timpului.

Algoritmul formal al metodei divide et impera:

funcţion *divimp*(*x*)
{returnează o soluţie pentru cazul *x*}
if *x* este suficient de mic **then return** *adhoc*(*x*)
{descompune *x* în subcazurile $x_1, x_2, \ldots, x_k$}
for $i \leftarrow 1$ **to** *k* **do** $y_i \leftarrow divimp(x_i)$
{recompune $y_1, y_2, \ldots, y_k$ în scopul obţinerii soluţiei *y* pentru *x*}
return *y*

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

Un algoritm 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"?

In exemplul precedent, cu toate ca valoarea lui n_0 nu influenţează ordinul timpului, este influenţata însă constanta multiplicativa a lui $n^{\lg 3}$, ceea ce poate avea un rol considerabil în eficienţa algoritmului. Pentru un algoritm divide et impera oarecare, chiar dacă ordinul timpului nu poate fi îmbunătăţit, se doreşte optimizarea acestui prag în sensul obţinerii unui algoritm cât mai eficient. Nu exista o metodă teoretică generală pentru aceasta, pragul optim depinzând nu numai de algoritmul în cauză, dar şi de

particularitatea implementării. Considerând o implementare dată, pragul optim poate fi determinat empiric, prin măsurarea timpului de execuţie pentru diferite valori ale lui n_0 şi cazuri de mărimi diferite.

In general, se recomandă o metodă hibridă care constă în a *a)* determinarea teoretică a formei ecuaţiilor recurente; *b)* găsirea empirică a valorilor constantelor folosite de aceste ecuaţii, în funcţie de implementare.

Revenind la exemplul nostru, pragul optim poate fi găsit rezolvând ecuaţia

$$t_A(n) = 3t_A(\lceil n/2 \rceil) + t(n)$$

Empiric, găsim $n_0 \cong 67$, adică valoarea pentru care nu mai are importanţă dacă aplicăm algoritmul *A* în mod direct, sau dacă continuăm descompunerea. Cu alte cuvinte, atâta timp cât subcazurile sunt mai mari decât n_0, este bine să continuăm descompunerea. Dacă continuăm însă descompunerea pentru subcazurile mai mici decât n_0, eficienţa algoritmului scade.

Observăm că metoda divide et impera este prin definiţie recursivă. Uneori este posibil să eliminăm recursivitatea printr-un ciclu iterativ. Implementată pe o maşina convenţionala, versiunea iterativă *poate fi* ceva mai rapidă (in limitele unei constante multiplicative). Un alt avantaj al versiunii iterative ar fi faptul că economiseşte spaţiul de memorie. Versiunea recursiva foloseşte o stivă necesară memorării apelurilor recursive. Pentru un caz de mărime *n*, numărul apelurilor recursive este de multe ori în $\Omega(\log n)$, uneori chiar în $\Omega(n)$.

1.1. Mergesort (sortarea prin interclasare)

Fie $T[1 .. n]$ un tablou pe care dorim sa-l sortam crescător. Prin tehnica divide et impera putem proceda astfel: separăm tabloul *T* în două părţi de mărimi cât mai apropiate, sortăm aceste părţi prin apeluri recursive, apoi interclasăm soluţiile pentru fiecare parte, fiind atenţi să păstrăm ordonarea crescătoare a elementelor. Obţinem următorul algoritm:

```
   procedure mergesort(T[1 .. n])
{sortează în ordine crescătoare tabloul T}
if n este mic
   then  insert(T)
   else  arrays U[1 .. n div 2], V[1 .. (n+1) div 2]
         U ← T[1 .. n div 2]
         V ← T[1 + (n div 2) .. n]
         mergesort(U); mergesort(V)
merge(T, U, V)
```

unde *insert*(*T*) este algoritmul de sortare prin inserţie cunoscut, iar *merge*(*T*, *U*, *V*) interclasează într-un singur tablou sortat *T* cele două tablouri deja sortate *U* şi *V*.

Algoritmul *mergesort* ilustrează perfect principiul divide et impera: pentru *n* având o valoare mica, nu este rentabil să apelăm recursiv procedura *mergesort*, ci este mai bine să efectuăm sortarea prin inserţie. Algoritmul *insert* lucrează foarte bine pentru $n \le 16$, cu toate că, pentru o valoare mai mare a lui *n*, devine neconvenabil. Evident, se poate concepe un algoritm mai puţin eficient, care să meargă pană la descompunerea totală; în acest caz, mărimea stivei este în $\Theta(\log n)$.

Spaţiul de memorie necesar pentru tablourile auxiliare *U* şi *V* este în $\Theta(n)$. Mai precis, pentru a sorta un tablou de $n = 2^k$ elemente, presupunând că descompunerea este totală, acest spaţiu este de

$$2(2^{k-1} + 2^{k-2} + \ldots + 2 + 1) = 2 \cdot 2^k = 2n$$

elemente.

In algoritmul *mergesort*, suma mărimilor subcazurilor este egală cu mărimea cazului iniţial. Această proprietate nu este în mod necesar valabilă pentru algoritmii divide et impera. Este esenţial ca subcazurile să fie de mărimi cât mai apropiate (sau, altfel spus, subcazurile să fie cât mai *echilibrate*).

1.2. Quicksort (sortarea rapida)

Algoritmul de sortare *quicksort*, inventat de Hoare în 1962, se bazează de asemenea pe principiul divide et impera. Spre deosebire de *mergesort*, partea nerecursivă a algoritmului este dedicata construirii subcazurilor şi nu combinării soluţiilor lor.

Ca prim pas, algoritmul alege un element *pivot* din tabloul care trebuie sortat. Tabloul este apoi partiţionat în două subtablouri, alcătuite de-o parte şi de alta a acestui pivot în următorul mod: elementele mai mari decât pivotul sunt mutate în dreapta pivotului, iar celelalte elemente sunt mutate în stânga pivotului. Acest mod de partiţionare este numit *pivotare*. În continuare, cele două subtablouri sunt sortate în mod independent prin apeluri recursive ale algoritmului. Rezultatul este tabloul complet sortat; nu mai este necesară nici o interclasare. Pentru a echilibra mărimea celor două subtablouri care se obţin la fiecare partiţionare, ar fi ideal să alegem ca pivot elementul median. Intuitiv, *mediana* unui tablou T este elementul m din T, astfel încât numărul elementelor din T mai mici decât m este egal cu numărul celor mai mari decât m. Din păcate, găsirea medianei necesita mai mult timp decât merită. De aceea, putem pur şi simplu să folosim ca pivot primul element al tabloului. Iată cum arată acest algoritm:

```
   procedure quicksort(T[i .. j])
{sortează în ordine crescătoare tabloul T[i .. j]}
if j–i este mic
   then insert(T[i .. j])
   else pivot(T[i .. j], l)
        {după pivotare, avem:
           i ≤ k < l  ⇒  T[k] ≤ T[l]
           l < k ≤ j  ⇒  T[k] > T[l]}
        quicksort(T[i .. l–1])
        quicksort(T[l+1 .. j])
```

Mai rămâne să concepem un algoritm de pivotare cu timp liniar, care să parcurgă tabloul T o singură dată. Putem folosi următoarea tehnica de pivotare: parcurgem tabloul T o singură dată, pornind însă din ambele capete. Încercaţi să înţelegeţi cum funcţionează acest algoritm de pivotare, în care $p = T[i]$ este elementul pivot:

```
   procedure pivot(T[i .. j], l)
{permută elementele din T[i .. j] astfel încât, în final,
 elementele lui T[i .. l–1] sunt ≤ p,
 T[l] = p,
 iar elementele lui T[l+1 .. j] sunt > p}
p ← T[i]
k ← i; l ← j+1
repeat  k ← k+1 until T[k] > p or k ≥ j
repeat  l ← l–1  until T[l] ≤ p
while k < l do
   interschimbă T[k] şi T[l]
   repeat  k ← k+1 until T[k] > p
   repeat  l ← l–1  until T[l] ≤ p
{pivotul este mutat în poziţia lui finală}
interschimbă T[i] şi T[l]
```

Intuitiv, ne dăm seama că algoritmul *quicksort* este ineficient, dacă se întâmpla în mod sistematic ca subcazurile $T[i \,..\, l–1]$ şi $T[l+1 \,..\, j]$ să fie puternic neechilibrate. Ne propunem în continuare să analizăm aceasta situaţie în mod riguros.

Operaţia de pivotare necesită un timp în $\Theta(n)$. Fie constanta n_0, astfel încât, în cazul cel mai nefavorabil, timpul pentru a sorta $n > n_0$ elemente prin *quicksort* să fie

$$t(n) \in \Theta(n) + \max\{t(i)+t(n–i–1) \mid 0 \le i \le n–1\}$$

Folosim metoda inducţiei constructive pentru a demonstra independent că $t \in O(n^2)$ şi $t \in \Omega(n^2)$.

Putem considera că există o constantă reală pozitivă c, astfel încât $t(i) \le ci^2+c/2$ pentru $0 \le i \le n_0$. Prin ipoteza inducţiei specificate parţial, presupunem că $t(i) \le ci^2+c/2$ pentru orice $0 \le i < n$. Demonstrăm că proprietatea este adevărata şi pentru n. Avem

$$t(n) \le dn + c + c \max\{i^2+(n-i-1)^2 \mid 0 \le i \le n-1\}$$

d fiind o altă constantă. Expresia $i^2+(n-i-1)^2$ îşi atinge maximul atunci când i este 0 sau $n-1$. Deci,

$$t(n) \le dn + c + c(n-1)^2 = cn^2 + c/2 + n(d-2c) + 3c/2$$

Dacă luăm $c \ge 2d$, obţinem $t(n) \le cn^2+c/2$. Am arătat că, dacă c este suficient de mare, atunci $t(n) \le cn^2+c/2$ pentru orice $n \ge 0$, adică, $t \in O(n^2)$. Analog se arată că $t \in \Omega(n^2)$.

Am arătat, totodată, care este cel mai nefavorabil caz: atunci când, la fiecare nivel de recursivitate, procedura *pivot* este apelata o singură dată. Dacă elementele lui T sunt distincte, cazul cel mai nefavorabil este atunci când iniţial tabloul este ordonat crescător sau descrescător, fiecare partiţionare fiind total neechilibrată. Pentru acest cel mai nefavorabil caz, am arătat că algoritmul *quicksort* necesită un timp în $\Theta(n^2)$.

Întrebări de control:

1. De ce această metodă se numeşte divide et impera ?
2. Ce este un algoritm recursiv?
3. Descrieţi etapele necesare pentru rezolvarea unei probleme prin metoda divide et impera.
4. Ce tip de funcţie descrie timpul de execuţie al unui algoritm de tipul divide et impera?
5. Care sunt avantajele şi dezavantajele algoritmilor divide et impera?