Laborator 1: Divide et Impera

Obiective laborator

- Înțelegerea conceptului teoretic din spatele descompunerii
- Rezolvarea de probleme abordabile folosind conceptul de Divide et Impera

Importanţă – aplicaţii practice

Paradigma Divide et Impera stă la baza construirii de algoritmi eficienți pentru diverse probleme:

- Sortări (ex: MergeSort [1] [http://www.sorting-algorithms.com/merge-sort], QuickSort [2] [http://www.sorting-algorithms.com/quick-sort])
- Înmulțirea numerelor mari (ex: Karatsuba [3] [http://en.wikipedia.org/wiki/Karatsuba_algorithm])
- Analiza sintactică (ex: parsere top-down [4] [http://en.wikipedia.org/wiki/Top-down_parser])
- Calcularea transformatei Fourier discretă (ex: FFT [5] [http://en.wikipedia.org/wiki/Fast_Fourier_transform])

Un alt domeniu de utilizare a tehnicii divide et impera este programarea paralelă pe mai multe procesoare, sub-problemele fiind executate pe mașini diferite.

Prezentarea generală a problemei

O descriere a tehnicii D&I: "Divide and Conquer algorithms break the problem into several sub-problems that are similar to the original problem but smaller in size, solve the sub-problems recursively, and then combine these solutions to create a solution to the original problem." [7]

Deci un algoritm D&I **împarte problema** în mai multe subprobleme similare cu problema inițială şi de dimensiuni mai mici, **rezolva sub-problemele** recursiv şi apoi **combina soluțiile** obţinute pentru a obține soluția problemei inițiale.

Sunt trei pași pentru aplicarea algoritmului D&I:

- **Divide**: împarte problema în una sau mai multe *probleme similare de dimensiuni mai mici*.
- **Impera** (stăpânește): rezolva subprobleme recursiv; dacă dimensiunea sub-problemelor este mica se rezolva iterativ.
- **Combină**: combină soluțiile sub-problemelor pentru a obține soluția problemei inițiale.

Complexitatea algoritmilor D&I se calculează după formula:

$T(n) = D(n) + S(n) + C(n)$,

unde $D(n)$, $S(n)$ şi $C(n)$ reprezintă complexitățile celor 3 pași descriși mai sus: divide, stăpânește respectiv combină.

Probleme clasice

1. Sortarea prin interclasare

Sortarea prin interclasare (MergeSort [1] [http://www.sorting-algorithms.com/merge-sort]) este un algoritm de sortare de vectori ce folosește paradigma D&I:

- **Divide**: împarte vectorul inițial în doi sub-vectori de dimensiune n/2.
- **Stăpânește**: sortează cei doi sub-vectori recursiv folosind sortarea prin interclasare; recursivitatea se oprește când dimensiunea unui sub-vector este 1 (deja sortat).
- **Combina**: Interclasează cei doi sub-vectori sortați pentru a obține vectorul inițial sortat.

Pseudocod:

```
MergeSort(v, start, end)                // v - vector, start - limită inferioară, end - limită superioară
        if (start == end) return;       // condiția de oprire
        mid = (start + end) / 2;        // etapa divide
        MergeSort(v, start, mid);       // etapa stăpânește
        MergeSort(v, mid+1, end);
        Merge(v, start, end);           // etapa combină

Merge(v, start, end)                    // interclasare sub-vectori
        mid = (start + end) / 2;
        i = start;
        j = mid + 1;
        k = 1;
        while (i <= mid && j <= end)
                if (v[i] <= v[j]) u[k++] = v[i++];
                else u[k++] = v[j++];

        while (i <= mid)
                u[k++] = v[i++];

        while (j <= end)
                u[k++] = v[j++];

        copy(v[start..end], u[1..k-1]);
```

Complexitatea algoritmului este dată de formula: T(n) = D(n) + S(n) + C(n), unde D(n)=O(1), S(n) = 2*T(n/2) și C(n) = O(n), rezulta T(n) = 2 * T(n/2) + O(n).

Folosind teorema Master [8] [http://people.csail.mit.edu/thies/6.046-web/master.pdf] găsim complexitatea algoritmului: **T(n) = O(n * lg n)**.

2. Căutarea binară

Se dă un **vector sortat crescător** (v[1..n]) ce conține valori reale distincte și o valoare x. Sa se găsească la ce poziție apare x în vectorul dat.

Pentru rezolvarea acestei probleme folosim un algoritm D&I:

- **Divide**: împărțim vectorul în doi sub-vectori de dimensiune n/2.
- **Stăpânește**: aplicăm algoritmul de căutare binară pe sub-vectorul care conține valoarea căutată.
- **Combină**: soluția sub-problemei devine soluția problemei inițiale, motiv pentru care nu mai este nevoie de etapa de combinare.

Pseudocod:

```
BinarySearch(v, start, end, x)
        if (start > end) return;        // condiția de oprire (x nu se află în v)
        mid = (start + end) / 2;        // etapa divide
        // etapa stăpânește
        if (v[mid] == x) return mid
        if (v[mid] > x) return BinarySearch(v, start, mid-1, x);
        if (v[mid] < x) return BinarySearch(v, mid+1, end, x);
```

Complexitatea algoritmului este data de relația T(n) = T(n/2) + O(1), ceea ce implica: **T(n) = O(lg n)**.

3. Turnurile din Hanoi

Se considera 3 tije A, B, C şi n discuri de dimensiuni distincte (1, 2.. n ordinea crescătoare a dimensiunilor) situate inițial toate pe tija A în ordinea 1,2..n (de la vârf către baza). Singura operație care se poate efectua este de a selecta un disc ce se află în vârful unei tije şi plasarea lui în vârful altei tije astfel încât să fie așezat deasupra unui disc de dimensiune mai mare decât a sa. Sa se găsească un algoritm prin care se mută toate discurile pe tija B (problema turnurilor din Hanoi).

Pentru rezolvarea problemei folosim următoarea strategie [9] [http://www.mathcs.org/java/programs/Hanoi/index.html]:

- mutam primele n-1 discuri de pe tija A pe tija C folosindu-ne de tija B.
- mutam discul n pe tija B.
- mutam apoi cele n-1 discuri de pe tija C pe tija B folosindu-ne de tija A.

Pseudocod [10]:

```
Hanoi(n, A, B, C)        // mută n discuri de pe tija A pe tija B fol tija C
        if (n >= 1)
                Hanoi(n-1, A, C, B);
                Muta_disc(A, B);
                Hanoi(n-1, C, B, A);
```

Complexitatea: T(n) = 2*T(n-1) + O(1), recurenta ce conduce la **$T(n) = O(2^n)$**.

Concluzii

Divide et impera este o tehnică folosită pentru a realiza algoritmi eficienți pentru diverse probleme. În cadrul acestei tehnici se disting trei etape: divide, stăpânește și combină.

Mai multe exemple de algoritmi care folosesc tehnica divide et impera puteți găsi la [11] [http://www.cs.berkeley.edu/~vazirani/algorithms/chap2.pdf].

Probleme laborator

Pentru punctaj maxim, **asistentul va alege** un subpunct pentru fiecare problemă.

Problema 1 [3p]

- **1.1** Se da un sir sortat. Gasiti numarul de elemente egale cu x din sir.

Exemplu: pentru sirul `{1 2 4 4 10 10 20}` si `x = 10`, x apare de 2 ori in sir.

- **1.2** Se da un numar natural n. Scrieti un algoritm de complexitate **O(log n)** care sa calculeze $\sqrt{n}$ cu o precizie de 0.001.

Exemplu: pentru 0.25 algoritmul poate da orice valoare intre 0.499 si 0.501 inclusiv.

Problema 2 [3p]

- **2.1** Fie o valoare întreagă necunoscută, pe care o denumim `unknown`. Gasiți valoarea lui `unknown` prin **Divide et Impera**, folosind metoda `isInBounds(int x)` pentru Java și `is_in_bounds(int x)` pentru C++ care întoarce:

  - true, dacă `x < = unknown`
  - false, dacă `x > unknown`

- **2.2** Se da un șir neordonat **S** cu `n - 1` numere distincte, selectate dintre cele `n` numere de la `0` la `n - 1`. Toate sunt numere întregi, reprezentate pe 32 de biti. Folosind metoda `getBit(int i, int j)` care intoarce al j-lea bit din reprezentarea binara a lui `S[i]`, determinati numarul lipsă.

Exemplu:

- pentru șirul {0 1 9 4 5 7 6 8 2} lipsește numarul 3
- `getBit(7,3)` întoarce al treilea bit din `S[7]`. S[7] este 8, în binar: 1000, deci bit-ul 3 este 1.
- La un prim pas, se observă că bitul 3 este 0 pentru doar 7 numere din șir, deși intre 0 si 9 sunt 8 numere care ar trebui să aiba bitul 3 egal cu 0, respectiv numerele de la 0 la 7. Prin urmare, la primul pas ne dam seama ca numarul cautat este intre 0 si 7.

Problema 3 [4p]

- **3.1 Statistici de ordine:** se dă un vector de numere întregi neordonate. Scriind o funcție de partitionare, folosiți **Divide et Impera** pentru
  - a determina a k-lea element ca mărime din vector
  - a sorta vectorii prin QuickSort

Exemplu: pentru vectorul `{0 1 2 4 5 7 6 8 9}`, al 3-lea element ca ordine este 2, iar vectorul sortat este {0 1 2 4 5 6 7 8 9}

- **3.2** Se da un sir **S** de n numere intregi. Sa se detemine cate inversiuni sunt in sirul dat. Numim inversiune o pereche de indici `1 ⇐ i < j ⇐ n` astfel incat `S[i] > S[j]`

Exemplu: in sirul `{0 1 9 4 5 7 6 8 2}` sunt 12 inversiuni.

- **3.3** Se dau `n - 1` numere naturale distincte intre `0` si `n - 1`. Scriind o functie de partitionare, determinati numarul lipsa.

Exemplu: pentru `n = 9` si vectorul `{0 1 9 4 5 7 6 8 2}`, numarul lipsa este `3`.

Referinţe

[1] MergeSort [http://www.sorting-algorithms.com/merge-sort]

[2] QuickSort [http://www.sorting-algorithms.com/quick-sort]

[3] Karatsuba [http://en.wikipedia.org/wiki/Karatsuba_algorithm]

[4] Top down parser [http://en.wikipedia.org/wiki/Top-down_parser]

[5] Fast Fourier Transform [http://en.wikipedia.org/wiki/Fast_Fourier_transform]

[6] Divide et impera [http://en.wikipedia.org/wiki/Divide_and_rule]

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms

[8] Teorema Master [http://people.csail.mit.edu/thies/6.046-web/master.pdf]

[9] Hanoi Applet [http://www.mathcs.org/java/programs/Hanoi/index.html]

[10] Cristian A. Giumale, Introducere in Analiza Algoritmilor (cap. 2.5.1)

[11] Chapter 2, Divide-and-conquer algorithms, Berkeley University
[http://www.cs.berkeley.edu/~vazirani/algorithms/chap2.pdf]

pa/laboratoare/laborator-01.txt · Last modified: 2013/02/23 21:13 by sorina.sandu