Algoritmi Divide-et-Impera

- una din tehnicile de baza pt designul algoritmilor
- reduce complexitatea: cel mai des de la n la log(n), de la n^2 la n log(n) etc

Structura unui algoritm divide-et-impera

- **divide:** imparte problema in (una sau mai multe) subprobleme de dimensiune mai mica (neaparat mai mica, recursivitatea trebuie sa ne conduca inspre problema elementara/banala)
- **impera:** rezolva recursiv aceste subprobleme sau iterativ daca am ajuns la problema elementara / banala => se obtin solutii pentru subprobleme
- **combina:** combina solutiile subproblemelor intr-o solutie pt intreaga problema

Utilizarea cea mai frecventa

- problema de dimensiune n se imparte in 2 subprobleme de dimensiune n/2
- se rezolva recursiv cele 2 subprobleme si solutiile se pot combina in timp liniar
- consecinta:
  - **forta bruta**: complexitate n^2
  - **divide-et-impera**: complexitate n log(n): $T(n) = 2T(n/2) + \Theta(n)$
- exemple: mergesort, quicksort

Abordarea designerului

- decide asupra unui numar de subprobleme (uzual 2 pt tablouri unidimensionale, 4 pt bidimensionale, etc)
- se intreaba: daca as cunoaste rezolvarile acestor subprobleme, as putea sa le combin intr-un timp superior rezolvarii problemei initiale prin forta bruta? care parte a impartirii mele face combinarea sa devina mai usoara decat problema initiala? (exemplu: daca pot rezolva sortarea in $\Theta(n^2)$ prin forta bruta, procedura merge din algoritmul mergesort trebuie sa fie mai rapida decat atat, altfel nu am castigat nimic)
- intotdeauna se intreaba si: *pot mai bine decat atat?*

Problema 1: numararea inversiunilor (NI)

- se da un sir $v_1, v_2 \ldots v_n$
- v_i si v_j formeaza o inversiune daca $i<j$ dar $v_i>v_j$
- exemplu: sirul {1,2,5,4,3} are 3 inversiuni: {5,4}, {5,3}, {4,3}
- aplicatii: metrica pt similaritate (exemplu: facem un top al melodiilor noastre preferate, numarul de inversiuni intre topurile noastre arata cat de mult se aseamana gusturile noastre), metrica pt ordonare (cu cat mai putine inversiuni are o lista, cu atat e mai ordonata), teoria voturilor, analiza sensibilitatii functiei google de ordonare a rezultatelor etc

NI – forta bruta

- prima idee este de a lua fiecare numar cu fiecare si a vedea daca formeaza o inversiune => complexitate $\Theta(n^2)$
- *pot mai bine decat atat?*

NI – divide-et-impera

- daca as cunoaste numarul de inversiuni din prima jumatate (inv1) si numarul de inversiuni din a 2ª jumatate (inv2), as putea obtine numarul de inversiuni (inv) din tot vectorul in timp liniar?
- observ ca inv = inv1+inv2+inv3, unde inv3 = numarul de inversiuni care apar intre elemente din prima jumatate si elemente din a 2ª jumatate; cum inv1 si inv2 se determina recursiv, problema devine daca pot determina inv3 in timp liniar
- practic un element din a 2ª jumatate formeaza o inversiune cu toate elementele din prima jumatate care sunt mai mari decat el; pot numara cate sunt acestea fara sa compar fiecare cu fiecare (pt ca asta mi-ar da complexitate patratica)?
- idee: mergesort! procedura merge stie, cand adauga un element din a 2ª jumatate, cate elemente au ramas neadaugate in prima jumatate (= cate sunt mai mari, adica exact cate inversiuni produce acel element) => nu mai e nevoie sa le compar fiecare cu fiecare, ci doar sa fac o adunare
- obs: fata de problema initiala, castig faptul ca in partea de combinare elementele din cele 2 jumatati sunt sortate, de aici si progresul de la patratic la liniar; daca nu castigam nimic, nu rezolvam nimic!
- complexitate: aceeasi recurenta ca la mergesort => $\Theta(n \log(n))$

```
NI(L)
    if L are 1 element
        return {0, L}
    divide L in 2 jumatati L1 si L2     // divide
    {inv1, L1} <- NI(L1)                // impera
    {inv2, L2} <- NI(L2)                // impera
    {inv3, L} <- mergeNI(L1,L2)         // combina
    return {inv1+inv2+inv3, L}          // combina

mergeNI(L1,L2)
    while L1,L2 nevide
        append elementul minim din L1 sau L2 la L si treci la elementul urmator
        if min era in L2
            inv <- inv+elemente-ramase-in-L1
    append restul listei nevide la L
    return {inv, L}
```

Problema 2: subsecventa de suma maxima (SSM)

- se da un sir de intregi $v_1, v_2 \ldots v_n$
- subsecventa de suma maxima din sir reprezinta valoarea maxima posibila pt $Suma_{k=i..j} v_k$
- cand toate numerele din sir sunt negative, rezultatul se considera 0
- exemple: {-2,**5,-1,3**,-2,1,-3} cu valoarea 7, {1,-2,**4,-1,-2,4**} cu valoarea 5

SSM – forta bruta

- prima idee este de a lua toti indicii de inceput si toti indicii de sfarsit posibili si de a calcula sumele dintre ei, retinand maximul => complexitate $\Theta(n^2)$ sau $\Theta(n^3)$, in functie de implementare
- *pot mai bine decat atat?*

SSM – divide-et-impera

- daca as cunoaste subsecventa de suma maxima din prima jumatate (s1) si subsecventa de suma maxima din a 2ª jumatate (s2), as putea obtine subsecventa de suma maxima (smax) din tot vectorul in timp liniar?
- observ ca smax = max(s1,s2,s3), unde s3 = subsecventa de suma maxima dintre cele care traverseaza obligatoriu atat prima cat si a 2ª jumatate; cum s1 si s2 se determina recursiv, problema devine daca pot determina s3 in timp liniar
- fata de problema initiala, in determinarea lui s3 am castigat faptul ca stim sigur ca 2 elemente fac parte din solutie: ultimul element din prima jumatate, respectiv primul element din a 2ª; fiecare jumatate avand un capat fixat (j e fix in prima jumatate, respectiv i in a 2ª), s3 se poate determina in timp liniar; tot ce trebuie sa fac este sa determin subsecventa de suma maxima care se termina obligatoriu cu v_{mijloc}, la fel pe cea care incepe obligatoriu cu $v_{mijloc+1}$, apoi sa le adun intre ele si sa obtin s3
- obs: strategia este similara cu cea de la problema anterioara: impartire in 2, determinarea unei formule pt obtinerea solutiei mari din subsolutii, observarea avantajelor pe care partea de combinare le are in raport cu problema initiala
- complexitate: $T(n) = 2T(n/2)+ \Theta(n) \Rightarrow \Theta(n \log(n))$

```
SSM(L)
    if L are 1 element
        return max(0,acel element)
    divide L in 2 jumatati L1 si L2          // divide
    s1 <- SSM(L1)                            // impera
    s2 <- SSM(L2)                            // impera
    s3 <- smax(L1,L2)                        // combina
    return max(s1,s2,s3)                     // combina

smax(L1,L2)
    scrt <- 0
    smax1 <- 0
    smax2 <- 0
    for elem = end(L1) to start(L1)
        scrt <- scrt+elem
        if smax1<scrt
            smax1 <- scrt
    scrt <- 0
    for elem = start(L2) to end(L2)
        scrt <- scrt+elem
        if smax2<scrt
            smax2 <- scrt
    return smax1+smax2
```

- *pot mai bine decat atat?*

SSM – liniar

- observ ca o solutie nu poate incepe cu un numar negativ
- mai mult decat atat, o solutie nu poate incepe cu un prefix negativ (o serie de numere cu suma negativa), intrucat partea care urmeaza ar avea suma mai mare singura, fara acest prefix

- privind din cealalta perspectiva, orice prefix pozitiv e “bun”: n-are sens sa incep solutia de la numarul urmator, pt ca acest prefix nu face decat sa o imbunatateasca
- concluzie: in permanenta tin un prefix curent si o suma maxima; cat timp prefixul curent e pozitiv il pastrez, e candidat la a fi prefixul unei noi sume maxime; cand devine negativ il arunc si o iau de la 0, am sanse mai bune asa
- obs: pt a mai scapa de un ciclu, a trebuit sa gasesc pe cale logica o serie de subsecvente care sigur nu pot fi solutii, si sa le elimin
- obs: orice prefix poate fi gandit in acelasi timp si in acelasi fel si ca un sufix: important e sa fie pozitiv
- complexitate $\Theta(n)$

```
SSM(L)
    scrt <- 0
    smax <- 0
    for elem = start(L) to end(L)
        scrt <- scrt+elem
        if scrt<0
            scrt <- 0
        if smax<scrt
            smax <-scrt
    return smax
```

Exercitiu: Problema 3: Ridicarea la putere a unui numar (x^n)

- forta bruta: complexitate $\Theta(n)$
- divide-et-impera: $T(n) = T(n/2)+ \Theta(1)$: complexitate $\Theta(\log(n))$

Problema 4: Numerele lui Fibonacci (FIB)

- sugestie: cautati pe google “fibonacci numbers in nature” pt niste detalii interesante privind aceste numere

FIB – varianta recursiva

- $T(n) = T(n-1)+T(n-2)+\Theta(1)$ => complexitate exponentiala
- **exercitiu**: desenati arborele de recurenta si deduceti limita inferioara ($2^{n/2}$) respectiv limita superioara (2^n) de complexitate
- **exercitiu**: deduceti exact de cate ori se apeleaza FIB(n), FIB(n-1), FIB(n-2) etc
- e rau sa calculez de atat de multe ori acelasi lucru; *pot mai bine decat atat?*

```
FIB(n)
    if n<2
        return 1
    else
        return FIB(n-1)+FIB(n-2)
```

FIB – varianta iterativa

- $T(n) = T(n-1)+ \Theta(1)$ => complexitate liniara
- *pot mai bine decat atat?*

```
FIB(n,a,b)
    if n=0
        return a
    else
```

```
return FIB(n-1,b,a+b)
```

FIB – divide-et-impera

- se pleaca de la observatia ca $((1\ 1)\ (1\ 0))^n = ((F_{n+1}\ F_n)\ (F_n\ F_{n-1}))$ (unde ((a b) (c d)) reprezinta o matrice cu prima linie (a b) si a doua linie (c d))
- **exercitiu**: sa se demonstreze aceasta proprietate prin inductie
- vom putea calcula FIB(n) in complexitate logaritmica folosind algoritmul divide-et-impera de ridicare la putere!
- **exercitiu pt acasa**: implementati cele 3 variante de algoritmi FIB si studiati diferentele de performanta

Pt cine vrea sa isi testeze suplimentar abilitatile in algoritmi divide-et-impera

- http://en.wikipedia.org/wiki/Closest_pair_of_points_problem