\

Determinarea complexitatii algoritmilor

- numararea este in general simpla pe algoritmi nerecursivi: identificam operatiile (critice) si de cate ori se executa acestea (v. insertion sort in laboratorul trecut)
- ceva mai dificil este cu algoritmii recursivi: nu este mereu clar sau usor de determinat cate apeluri recursive se executa (si fiecare are alta talie)

Determinarea complexitatii unui algoritm recursiv

costul recursiv al unui apel

- cate **noi apeluri recursive** lanseaza un anumit apel
- **dimensiunea parametrilor** acestor noi apeluri

costul nerecursiv al unui apel

- orice altceva mai face un apel in afara de a lansa noi apeluri recursive

T(n) = cost_recursiv(n) + cost_nerecursiv(n)

Relatii de recurenta uzuale

divide et impera

- problema e impartita in **b subprobleme** de **dimensiune n/c**
- b si c difera de la un algoritm la altul, cel mai des b=c=2
- $T(n) = bT(\frac{n}{c}) + f(n)$

chip and conquer

- in fiecare pas, problema este redusa la **o subproblema** de **dimensiune n-c** (practic se elimina c posibilitati), cu ajutorul unor operatii care vor constitui cost nerecursiv (f(n))
- $T(n) = T(n-c) + f(n)$

chip and be conquered

- **b subprobleme** de **dimensiune n-c**
- $T(n) = bT(n-c) + f(n)$

Exercitiu: Dati exemple de algoritmi care au recurente de tipul celor de mai sus.

Determinarea recurentei de complexitate pt un algoritm recursiv: mergesort

Algoritmul de sortare prin interclasare (mergesort, in engleza) sorteaza unui vector de n elemente printr-o strategie divide et impera:

- imparte vectorul in 2 jumatati (**divide**)
- sorteaza fiecare jumatate folosind sortare prin interclasare (**impera**)
- combina cei 2 vectori sortati intr-un singur vector mare, sortat (**combina**)

```
mergesort(start, stop)
{
  if start<stop then
  {
    mijloc=(start+stop)/2
    mergesort(start, mijloc)  // sorteaza prima jumatate
```

```
    mergesort(mijloc+1, stop)  // sorteaza a doua jumatate
    merge(start, mijloc, stop)  // interclaseaza vectorii sortati
  }
}

merge(start, mijloc, stop)
{
  for i=start to stop
    b[i]=a[i] // copiaza jumatatile sortate intr-un vector auxiliar b

  i=start; j=mijloc+1; k=start;

  // copiaza inapoi in a pe cel mai mic dintre elementele curente in
  // cele 2 jumatati
  while (i<=mijloc) and (j<=stop)
    if b[i]<=b[j] then
      a[k++]=b[i++]
    else
      a[k++]=b[j++]

  // copiaza ce a mai ramas din prima jumatate, daca in ea a mai ramas
  while i<=mijloc
    a[k++]=b[i++]
  // similar pt a doua
  while j<=stop
    a[k++]=b[j++]
}
```

Exercitii:
Care este costul nerecursiv?
Raspuns: Costul nerecursiv este efortul efectuat de functia merge. Aici se asaza pe rand n elemente intr-un nou vector sortat. Pentru a alege al i-lea element, se realizeaza maxim 3 teste (i<=mijloc, j<=stop, b[i]<=b[j]) si 3 atribuiri (a[k]=b[i], k++, i++), prin urmare costul nerecursiv este Θ(n). Operatia de copiere care precede interclasarea propriu-zisa este in mod evident si ea Θ(n) si nu afecteaza complexitatea functiei.

⇨ **f(n) = Θ(n)**

Care este costul recursiv?
Raspuns: Ne uitam in functia mergesort, unde se fac 2 apeluri recursive, fiecare de dimensiune n/2 (fiecare din ele pe un vector de 2 ori mai mic decat cel din pasul curent).

⇨ **cost_recursiv(n) = 2T(n/2)**

Se obtine recurenta de complexitate:
T(n) = 2T(n/2) + Θ(n)

Obs: O recurenta de complexitate are nevoie si de un caz de baza (orice recursivitate are o iesire, cand problema devine elementara). Cazul de baza spune care este complexitatea rezolvarii problemei elementare. Aici: **T(1)= Θ(1)**, cu semnificatia “atunci cand am de sortat vectori de 1 sg element, pot face asta intr-un numar constant de operatii”.

Metode de rezolvare a recurentelor de complexitate

- nu exista o metoda universal valabila
- este necesara cunoasterea mai multor metode, fiecare cu aria ei de aplicabilitate (similar cu metodele existente pt rezolvarea integralelor)

Metoda iterativa: presupune sa calculezi suma seriei rezultata din eliminarea recurentei
Exemplu (mergesort):
T(n) = 2T(n/2) + Θ(n), T(1) = Θ(1)

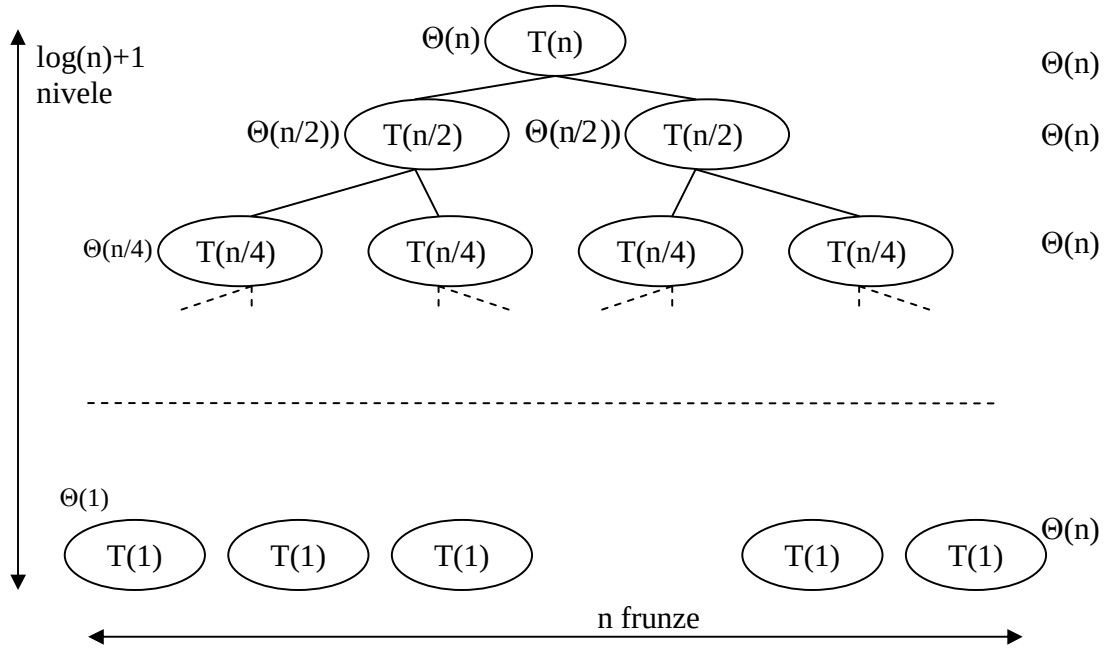
$$2^0 T(n) = 2^1 T(\frac{n}{2^1}) + 2^0 \theta(n)$$
$$2^1 T(\frac{n}{2^1}) = 2^2 T(\frac{n}{2^2}) + 2^1 \theta(\frac{n}{2^1})$$
$$2^2 T(\frac{n}{2^2}) = 2^3 T(\frac{n}{2^3}) + 2^2 \theta(\frac{n}{2^2})$$
.....
$$2^q T(\frac{n}{2^q}) = 2^{q+1} T(\frac{n}{2^{q+1}}) + 2^q \theta(\frac{n}{2^q})$$

Recurenta se termina atunci cand se ajunge la T(1) (pentru care cunoastem complexitatea, Θ(1)), deci cand $n/2^q=1$ => q=log(n).
Adunand termenii stangi, respectiv termenii drepti din toate aceste ecuatii si reducand termenii care apar atat in stanga, cat si in dreapta, obtinem:

$$T(n) = \sum_{q=0}^{\log(n)} 2^q \theta(\frac{n}{2^q}) = \sum_{q=0}^{\log(n)} \theta(n) = \theta(n)(1+\log(n))$$

Reprezentarea cu arbori de recurenta

- **nivel** in arbore = nivel de adancime in recursivitate
- **nod** = apel recursiv
- pe fiecare nivel este figurat si costul nerecursiv platit la acel nivel
- la fel este figurat costul nerecursiv in dreptul unui nod



Exemplu (mergesort): rezulta log(n)+1 nivele in arbore, pe fiecare executandu-se un efort de Θ(n). Asadar:
T(n) = (1+log(n))Θ(n) = Θ(n+n log(n)) = Θ(n log(n)).

Metoda substitutiei

- intr-un anumit sens, aceasta metoda este universal valabila
- problema este ca trebuie sa incepi prin a **ghici** o solutie F(n) pentru recurenta de complexitate
- apoi iti confirmi intuitia prin **inductie** dupa n
- in general se foloseste una din metodele mai putin riguroase pt a ghici o solutie, apoi metoda substitutiei pt a demonstra corectitudinea solutiei ghicite
- pentru recurenta **T(n) = f(T(g(n)))** (neaparat **g(n)<n**), cu **$T(n_0)$=c** pe cazul elementar, si pentru "ghiceala" F(n), schema metodei substitutiei este urmatoarea:

$$\frac{\text{caz de baza} : F(n_0) = c \qquad \text{pas de inductie} : \dfrac{\text{ipoteza inductiva} : T(g(n)) = F(g(n))}{T(n) = F(n)}}{\forall n \geq n_0 \,|\, (\exists n' \in \mathbf{N} \bullet n = g(n')) \bullet T(n) = F(n)}$$

Schema in cuvinte

- identificam n_0, c, g(n) si alegem F pentru recurenta de care ne ocupam
- verificam cazul de baza ($F(n_0)$=c)
- demonstram ca T(g(n))=F(g(n)) => T(n)=F(n)

Exemplu (mergesort):
T(n) = 2T(n/2) + Θ(n), T(1) = Θ(1)

Identificam constantele si alegem solutia: n_0=1, c=Θ(1), g(n)=n/2, F(n)=Θ(n+n log(n))

Caz de baza:
$T(n_0) = F(n_0)$ <=> T(1)=F(1)=Θ(1) --- OK

Pas de inductie:
Ipoteza inductiva: T(n/2) = F(n/2) = Θ(n/2+n/2 log(n/2)). **Trebuie aratat:** T(n) = F(n).

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

Problema: incercam sa demonstram prin inductie ca n=O(1).

Caz de baza: 1=O(1) --- OK

Pas de inductie:
Ipoteza inductiva: n-1=O(1). **Trebuie aratat:** n=O(1).
n = n-1+1 = O(1)+O(1) = O(1) --- OK

Exercitiu: Ce e in neregula in demonstratia de mai sus?
Raspuns: Se schimba constanta din definitia lui O pe parcursul demonstratiei!

Obs: Nu este riguros sa umblam cu O, Ω, Θ in inductii, tocmai din cauza ca riscam sa schimbam constantele si demonstratiile sa nu mai fie corecte. Trebuie, in loc, sa umblam chiar cu constantele din definitiile notatiilor de complexitate.

Metoda substitutiei aplicata riguros pt mergesort:
$T(n) = 2T(n/2) + \Theta(n)$, $T(1) = \Theta(1)$

Identificam constantele si alegem solutia: $n_0=1$, $c=\Theta(1)$, $g(n)=n/2$, $F(n)=\Theta(n+n \log(n))$.
Adica: $c_1(n+n \log(n)) \leq F(n) \leq c_2 (n+n \log(n))$.

Caz de baza:
$T(n_0) = F(n_0) <=> T(1)=F(1)=\Theta(1)$ --- OK

Pas de inductie:
Ipoteza inductiva: $c_1 (n/2+n/2 \log(n/2)) \leq T(n/2) \leq c_2 (n/2+n/2 \log(n/2))$.
Trebuie aratat: $c_1 (n+n \log(n)) \leq T(n) \leq c_2 (n+n \log(n))$.

$T(n) = 2T(n/2) + \Theta(n)$
$c_1 (n+n \log(n/2)) + c_3 n \leq T(n) \leq c_2 (n+n \log(n/2)) + c_4 n$
$c_1 (n+n (\log(n)-1)) + c_3 n \leq T(n) \leq c_2 (n+n(\log(n)-1)) + c_4 n$
$c_1 n \log(n) + c_3 n \leq T(n) \leq c_2 n \log(n) + c_4 n$

Daca alegem de la inceput $c_1<c_3$ si $c_2>c_4$ (permis, intrucat c_3 si c_4 sunt fixate):
$c_1 (n+n \log(n)) \leq T(n) \leq c_2 (n+n \log(n))$ --- OK

Exercitiu: refaceti demonstratia pt n=O(1) prin metoda riguroasa.

Metoda master

- un set de criterii pt a determina comportamentul algoritmilor divide et impera
- valabila pt algoritmi care au recurenta de complexitate de forma $T(n) = bT(n/c) + f(n)$ si respecta, in plus, anumite restrictii

Schita de justificare a teoremei master
Pt recurenta de mai sus, arborele de recurenta va contine pe al doilea nivel b noduri (fiecare nod reprezentand o problema de dimensiune n/c), pe urmatorul b^2, apoi tot asa pana la ultimul nivel pe care se vor gasi $b^{\text{inaltimea\_arborelui}}$ frunze.

$\Rightarrow nr\_frunze = b^{\log_c(n)} = n^{\log_c(b)} = n^{\frac{\log(b)}{\log(c)}} = n^E$ (*notatie*)

Considerand doar varful si baza arborelui de recurenta:

- la **varf**, costul semnificativ este costul nerecursiv **f(n)**
- la **baza**, fiecare frunza rezolva o problema elementara, iar costul nivelului ultim din arbore e influentat nu atat de ce face fiecare frunza, cat de numarul de frunze, $\mathbf{n^E}$, care determina de cate ori se face acel efort "elementar"
- intrebarea devine care dintre costul de la varf si cel de la baza este dominant; daca este unul dominant, atunci el va dicta complexitatea algoritmului

Cele 3 cazuri din teorema master se bazeaza pe relatia dintre aceste 2 costuri, in felul urmator:

Cazul 1 (n^E dominant)

- valabil daca $\exists\ \varepsilon>0$ a.i. $f(n)\in O(n^{E-\varepsilon})$
- $T(n) = \Theta(n^E)$

Cazul 2 (costuri similare pentru n^E si f(n))

- valabil daca $f(n)\in \Theta(n^E)$
- in acest caz, efortul e distribuit uniform intre nivelele din arbore si va fi de ordinul n^E * numarul_de_nivele
- $T(n) = \Theta(n^E \log(n))$

Cazul 3 (f(n) dominant)

- valabil daca $\exists\ \varepsilon>0$ a.i. $f(n)\in \Omega(n^{E+\varepsilon})$ si, in plus, $\exists$ constantele $0<a<1$, n_0, a.i. pt orice $n\geq n_0$, b $f(n/c) \leq$ a f(n)
- $T(n) = \Theta(f(n))$
- daca nu s-ar indeplini conditia suplimentara b $f(n/c) \leq$ a f(n), costul nerecursiv ar creste foarte mult in josul arborelui, iar f(n) nu ar mai fi dominant

Exercitii:

$T(n) = 2T(n/2) + \Theta(n)$ (mergesort)

$T(n) = 3T(n/2) + \Theta(n)$

$T(n) = 3T(n/3) + n^2$

$T(n) = 2T(n/2) + n \log(n)$

$T(n) = 4T(n/2) + n$