Eficienta algoritmilor

În cursul de introducere am menționat că, oricât de rapid ar deveni un calculator, sau oricât de mult s-ar ieftini memoria, eficiența va fi un factor decisiv în alegerea unui algoritm. Iată mai jos un exemplu pentru această afirmație.

Căutare secventială si căutare binară

Problema căutării sună astfel: se dă un șir de elemente sortate, și se cere căutarea unei valori în șir, respectiv returnarea poziției acelei valori din șir, în cazul în care aceasta se află în acel șir.

Prima abordare și cea mai simplă ar fi să parcurgem tot șirul astfel:

$SEQSEARCH(S, x)$

1. $location \leftarrow 1$
2. $\boldsymbol{while}\ (location \leq length[S]\ AND\ S[location] \neq x)\ \boldsymbol{do}$
3. $\quad location \leftarrow location + 1$
4. $\boldsymbol{if}\ (\ location\ > n\ )\ \boldsymbol{then}$
5. $\quad location \leftarrow -1$
6. $\boldsymbol{return}\ location$

În momentul în care găsim valoarea căutată, se părăsește bulca repetitivă, și se returnează locația găsită, altfel în caz că valoare nu este găsită, returnăm o poziție eronată și anume -1.

Un alt algoritm de căutare este cel descris mai jos și funcționează astfel: se împarte șirul în două și se compară valoarea de căutat cu cea din mijlocul șirului. Dacă nu este găsită, pasul de mai sus se va repeta în șirul din stânga, dacă valoarea din șir este mai mare decât cheia de căutat, respectiv în dreapta în caz contrar. Procedura este repetată până când fie valoarea este găsită, fie nu se mai poate împărți subșirul mai departe, caz în care concluzionăm că valoarea nu este în șir.

$BINSEARCH(S, x)$

1. $low \leftarrow 1$
2. $high \leftarrow length[S]$
3. $location \leftarrow 0$
4. $\boldsymbol{while}$ ($low \leftarrow high\ AND\ location = 0$) $\boldsymbol{do}$
5. $\quad mid = (\ low\ + high\ )/2$
6. $\quad \boldsymbol{if}\ (x = S[mid])\ \boldsymbol{then}\ location = mid$
7. $\quad \boldsymbol{else\ if}\ (\ x\ < S[mid])\ \boldsymbol{then}$
8. $\quad\quad high = mid - 1$
9. $\quad \boldsymbol{else}$
10. $\quad\quad low = mid + 1$

Pentru a compara cele două sortări, să presupunem pentru început că S conține 32 de elemente și x nu este în șir. Primul algoritm va compara toate cele 32 de elemente pentru a determina că x nu se află în șir. În general, căutarea secvențială va efectua n comparații, pentru a determina că x nu se află în șir.

Apoi să consideră, algoritmul $BINSEARCH$, ce efectuează două comparații ale lui x cu $S[mid]$. Într-un program optimizat, x ar fi comparat cu $S[mid]$ și rezultatul păstrat, așa că putem presupune că doar o singură comparație se va face la fiecare execuție a lui $while$. Dacă x ar fi mai mare decât toate elementele din șir s-ar executa *5+1* comparații. De remarcat că $6 = \log_2 32 + 1$. Pentru un n oarecare, numărul de comparații va fi $\log_2 n$ +1.

Comparațiile se vor face ca în figura de mai jos:

Figura 1. Căutarea binară într-un șir de 32 de elemente, atunci când valoarea este mai mare decât orice număr din șir.

Tabelul de mai jos prezintă numărul de comparații efectuate în căutarea secvențială și cea binară pentru diverse valori ale lui n, numărul de elemente din șir.

Mărimea șirului	Numărul de comparații în căutarea secvențială	Numărul de comparații în căutarea binară
128	128	8
1024	1024	11
1048576	1048576	21
4294967296	4294967296	33

Din tabelul de mai sus, este evident că alegerea unui algoritm eficient, este de dorit și înseamnă o îmbunătățire a timpului și nu numai. Alte exemple pot fi date, în continuare vom încerca să deprindem modul de a analiza eficiența unui algoritm.

Analiza algoritmilor

Pentru a determina eficiența unui algoritm, acesta trebuie analizat. Comparația de mai sus este una informală, așa că pentru a putea afirma ceva despre un algoritm trebuie să efectuăm ceva mai mulți pași.

Complexitatea algoritmilor

Atunci când analizăm eficiența unui algoritm, nu determinăm numărul de cicluri CPU necesare, și nu suntem interesați de frecvența memoriei sau alte date legate de arhitectura a calculatorului specifică pe care va rula algoritmului. Am văzut că algoritmul de căutare binară este mult mai eficient decât cel secvențial prin numărul de comparații, mult mai mare în cazul celui de-al doilea, pentru un șir cu n elemente. Aceasta este o tehnică de a analiza algoritmi. În general, timpul de rulare crește dacă și numărul de elemente de la intrare crește. De aceea analizăm eficiența unui algoritm determinând numărul de operații elementare necesare îndeplinirii sarcinii, unde acest număr este o funcție de *mărimea intrării*.

Pentru mulți algoritmi este ușor să găsim o măsură a mărimii de intrare. De exemplu, să luăm în considerare algoritmii de mai sus. n, este numărul de elemente din șir, și este o măsură simplă a mărimii de intrare. Vom denumi n, mărimea intrării. În cazul înmulțirii a două matrice, $m \times n$, numărul de rânduri și coloane reprezintă mărimea intrării. În unii algoritmi avem nevoie de două mărimi de intrare,

de exemplu dacă intrarea este un graf, vom avea ca măsură de intrare numărul de muchii și noduri. Uneori, trebuie să avem grijă ce anume denumim mărime de intrare. De exemplu, în algoritmul lui Fibonacci, se poate considera n mărimea de intrare, unde n este ordinul termenului Fibonacci. Totuși n este intrarea, nicidecum mărimea intrării. Pentru acest algoritm, dacă dorim să respectăm rigurozitatea, o măsură a mărimii de intrare este numărul de biți pentru a codifica ordinul termenului, care este $\log_2 n + 1$. De exemplu :

$$n = 13 = \underbrace{1101}_{4 \text{ bits}}{}_2$$

De aceea mărimea pentru $n = 13$ este 4.

După ce determinăm mărimea de intrare, vom calcula *complexitatea* algoritmului prin calculul numărului de *operații de bază*, sau elementare, necesare efectuării algoritmului.

Uneori, vom considera, pentru ușurința calculului, o operație de bază simplă cum este adunarea și alteori o operație de bază mai complexă, compusă din operații simple. De exemplu, într-un algoritm ce sortează prin compararea cheilor, vom considera instrucțiunea de comparație o operație de bază și asignarea o altă operație de bază. Astfel putem separa, și calcula cu o acuratețe mai mare eficiența acelui algoritm. Alteori, când dorim o estimare a eficienței algoritmului, putem cupla cele două operații într-una singură și denumi acea operație de *bază*.

În multe cazuri o operație de bază este efectuată în același mod de un număr de *n* ori. În aceste cazuri, $T(n)$ este definit ca numărul de ori de câte ori algoritmul efectuează o operație de bază, pentru o instanță de mărime de intrare n. $T(n)$ este denumită *complexitate de timp valabilă în orice caz*, iar determinarea lui $T(n)$ se numește *analiza complexității de timp valabilă în orice caz.*

Pentru a putea înțelege mai ușor analiza algoritmilor este indicat să facem o incursiune în matematică, mai exact în analiza creșterii funcțiilor.

Creșterea functiilor

Ordinul de timp, descris în primul curs, oferă o caracterizare simplă a eficienței algoritmului și permite compararea unui algoritm cu altul. Atunci când mărimea intrării este suficient de mare, pentru ca ordinul de timp să fie relevant, studiem eficiența *asimptotică* a algoritmilor. De obicei, un algoritm care este asimptotic mai eficient, este alegerea potrivită, excepție făcând cazurile în care mărimea de intrare este foarte mică.

Notatia asimptotică

Notațiile folosite pentru descrierea timpului de rulare, sunt definite ca funcții a căror domenii sunt mulțimi de numere naturale $N = \{0,1,2, \dots\}$. Notația asimptotică poate fi extinsă pe un domeniu de numere reale, dacă acest lucru este necesar.

Notatia Θ

În primul curs, am găsit că $T(n) = \Theta(n^2)$. Să definim semnificația notației. Pentru o funcție dată $g(n)$, vom nota $\Theta(g(n))$ cu setul de funcții care:

$$\Theta(g(n)) = \left\{ f(n) \middle| \begin{matrix} există\ constante\ pozitive\ c_1, c_2\ și\ n_0 \\ astfel\ ca\ 0 \le c_1 g(n) \le f(n) \le c_2 g(n)\ pentru\ orice\ \ n \ge n_0 \end{matrix} \right\}$$

O funcție $f(n)$ aparține mulțimii $\Theta(g(n))$ dacă există constante pozitive c_1 și c_2 astfel încât ea poate fii delimitată de funcțiile $c_1 g(n)$ și $c_2 g(n)$, pentru un n suficient de mare. Deși $\Theta(g(n))$ este o mulțime, vom scrie "$f(n) = \Theta(g(n))$" pentru a indica faptul că $f(n)$ este un membru al $\Theta(g(n))$.

În figura 2 sunt reprezentate funcțiile $f(n)$ și $g(n)$ unde $f(n) = \Theta(g(n))$. Pentru toate valorile n mai mari ca n_0, valorile lui $f(n)$ se găsesc între $c_1 g(n)$ și $c_2 g(n)$. Vom spune că $g(n)$ este *asimptotic strâns legată* de $f(n)$.

Definiția lui $\Theta(g(n))$ presupune ca fiecare membru al lui $\Theta(g(n))$ să fie asimptotic nenegativ, adică $f(n)$ să fie nenegativă când n este suficient de mare.

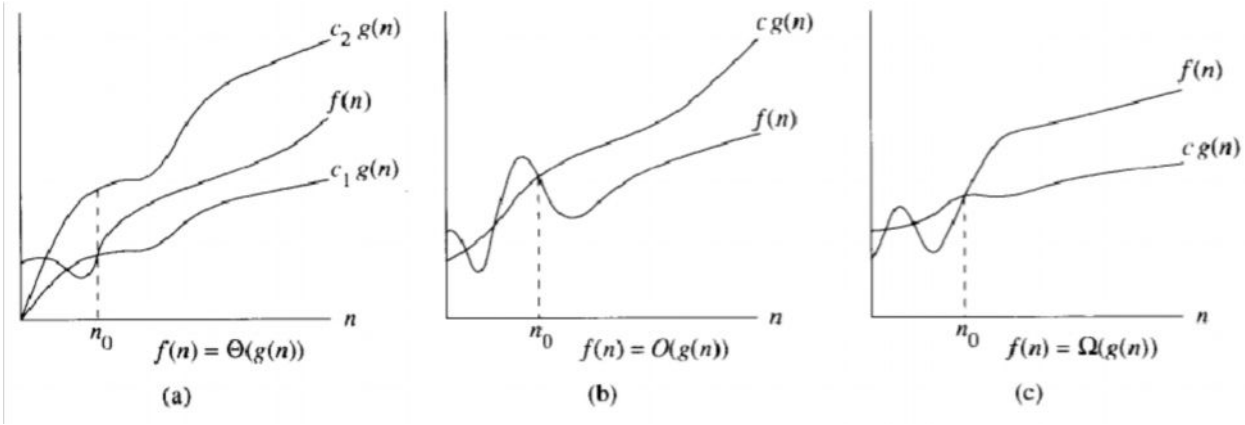

Figura 2. Exemple grafice ale notațiilor Θ, O, Ω. În fiecare parte, valoarea lui n_0 este valoarea minimă posibilă. a) Notația Θ care *strânge* funcția între două constante. a) notația O ce mărginește superior funcția cu un factor constant. c) notația Ω ce mărginește inferior funcția printr-un factor constant. Pentru un $n > n_0$, valoarea lui $f(n)$ va fi deasupra lui $cg(n)$.

Pentru a exemplifica notația asimptotică $\Theta(g(n))$, să demonstrăm că $\frac{1}{2}n^2 - 3n = \Theta(n^2)$. Pentru a realiza acest lucru. Trebuie să determinăm constantele *pozitive* c_1, c_2 și n_0 astfel ca:

$c_2 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$ pentru orice $n \geq n_0$. Împărțim prin n^2, rezultă:

$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$.

Inegalitatea din dreapta, poate fi realizată pentru $n \geq 1$ alegând $c_2 \geq \frac{1}{2}$. De asemenea, inegalitatea din dreapta poate fi îndeplinită, alegând $n \geq 7$ și alegând $c_1 \leq \frac{1}{14}$.

Astfel alegând $c_1 = \frac{1}{14}$, $c_2 = \frac{1}{2}$ și $n_0 = 7$, putem verifica că $\frac{1}{2}n^2 - 3n = \Theta(n^2)$.

Evident, se pot alege alte constante, important este că am găsit una din opțiuni.

Putem folosi de asemenea, definiția formală pentru a verifica dacă $6n^3 \neq \Theta(n^2)$. Să presupunem, pentru a crea o contradicție, că există c_2 și n_0 astfel încât $6n^3 \leq c_2 n^2$ pentru orice $n \geq n_0$. Dar atunci, $n \leq \frac{c_2}{6}$ ceea ce nu este posibil pentru un n oricât de mare, deoarece c_2 este constant. Intuitiv, termenii de grad mai mic ai funcției asimptotic pozitivă pot fi ignorați deoarece ei devin nesemnificativi pentru un n foarte mare. De aceea, setând c_1 la o valoare puțin mai mică decât coeficientul celui mai mare termen, și c_2 la o valoare puțin mai mare decât coeficientul celui mai mare termen permite ca inegalitatea din definiția notației Θ să fie satisfăcută.

Un alt exemplu, pornind de la funcția de grad doi, $f(n) = an^2 + bn + c$, unde a, b, c sunt constante cu $a > 0$, eliminând termenii de grad inferior obținem $f(n) = \Theta(n^2)$.

Urmărind pașii de mai sus, alegem constantele $c_1 = \frac{a}{3}$, $c_2 = \frac{5a}{3}$ și $n_0 = 2 \cdot max\left(\left(\frac{|b|}{a}\right), \left(\sqrt{\frac{|c|}{a}}\right)\right)$.

Se poate verifica, că $0 \leq c_1 n^2 \leq an^2 + bn + c \leq c_2 n^2$ pentru orice $n \geq n_0$. În general pentru orice funcție polinomială în care termenul cu cel mai mare grad are coeficient pozitiv avem $p(n) = \Theta(n^d)$ unde d este gradul polinomului.

Notatia O

Pentru a defini o limită asimptotic superioară a unei funcții folosim notația O. Pentru o funcție dată, $g(n)$, notăm cu $O(g(n))$ un set de funcții astfel:

$$O(g(n)) = \left\{ f(n) \middle| \begin{array}{c} există\ constamtele\ pozitive\ c\ și\ n_0\ astfel\ ca \\ 0 \leq f(n) \leq cg(n)\ pentru\ orice\ n \geq n_0 \end{array} \right\}$$

Vom folosi notația O pentru a desemna limita superioară a unei funcții. Figura 2b) prezintă acest lucru. Pentru orice $n > n_0$, valoarea funcției $f(n)$ este mai mică sau cel mult egală cu $g(n)$.

Pentru a indica faptul că o funcție $f(n)$ este membră a $O(g(n))$ vom scrie $f(n) = O(g(n))$. De notat că $\Theta(g(n)) \subseteq O(g(n))$. Demonstrația că orice funcție de grad 2 cu $a > 0$ este în $\Theta(n^2)$ este valabilă și în cazul notației O, ceea ce înseamnă că orice funcție de grad 2 cu $a > 0$ se află în $O(n^2)$.

Ce este mai important, este că orice funcție lineară $an + b$ se află în $O(n^2)$, afirmație ce se verifică simplu dacă $c = a + |b|$ și $n_0 = 1$. Această afirmație poate pare ciudată, dacă luăm cazul $n = O(n^2)$. Notația O este folosită , mai ales pentru a descrie limitele asimptotic superioare, în sensul în care și Θ realizează, pe de o parte același lucru. Totuși, nu specificăm *cât* de aproape de funcția originală $f(n)$ se găsește funcția ce o delimitează.

Folosind notația $O(n)$, descriem deseori timpul de rulare al algoritmului. De exemplu o structură repetitivă în altă structură repetitivă (denumită și imbricată în altă structură repetitivă), va avea $O(n^2)$ în cel mai rău caz. Costul buclei interioare variază de la $O(1)$ (constant) la n.

Din moment ce notația O descrie limita superioară, o vom folosi pentru a descrie cel mai rău caz, astfel $O(n^2)$ este o descriere corectă a celui mai nefavorabil caz pentru algoritmul de mai sus. Faptul că $O(n^2)$ este limita superioară, nu implică faptul că algoritmul este cuprins de limitele date de $\Theta(n^2)$. De exemplu, pentru algoritmul de inserție când șirul este deja sortat obținem un $\Theta(n)$ suficient pentru a delimita timpul necesar rulării algoritmului.

Notatia Ω

Această notație oferă un mod de a exprima *limita asimptotic inferioară* a unei funcții. Pentru o funcție dată $g(n)$, vom nota cu $\Omega(g(n))$, setul de funcții:

$$\Omega\big(g(n)\big) = \left\{ f(n) \middle| \begin{array}{c} exist\check{a}\ constante\ pozitive\ c\ \text{şi}\ n_0\ astfel\ ca \\ 0 \le cg(n) \le f(n)\ pentru\ orice\ \ n \ge n_0 \end{array} \right\}$$

Figura 2c) prezintă această mulțime de funcții.

Teorema 1

Pentru orice două funcții $f(n)$ și $g(n)$, $f(n) = \Theta(g(n))$ dacă și numai dacă $f(n) = O(g(n))$ și $f(n) = \Omega(g(n))$.

Ca un exemplu pentru această teoremă, să considerăm că $an^2 + bn + c = \Theta(n^2)$ pentru orice constante a, b, c cu $a > 0$. Aceasta implică faptul că $an^2 + bn + c = O(n^2)$ și că $an^2 + bn + c = \Omega(n^2)$.

Din moment ce Ω descrie o limită inferioară, o vom folosi pentru a descrie cel mai bun caz al unui algoritm. Să luăm spre exemplu, sortarea prin inserție.

Timpul de rulare pentru algoritmul acesta, se află între $\Omega(n)$ și $O(n^2)$ din moment ce se află între o funcție liniară și una pătratică. Mai mult, aceste legături sunt asimptotic precise, adică timpul de rulare nu este în $\Omega(n^2)$ din moment ce în cel mai bun caz algoritmul nu rulează decât în timp n.

Notația asimptotică definește o relație de ordine parțială între funcții, adică între eficiența relativă a diferiților algoritmi care rezolvă o anumită problemă: pentru oricare două funcții $f, g : N \to R^*$ definim relația *binară* : $f \leq g$ dacă $O(f) \subseteq O(g)$. Relația " $\leq$ " este o relație de *ordine parțială* în mulțimea funcțiilor definite pe N cu valori în R^*. Definim și o relație de echivalență $f \equiv g$ dacă $O(f) = O(g)$.

În mulțimea $O(f)$ putem înlocui pe f cu orice altă funcție echivalentă cu f. De exemplu $\log n \equiv \ln n$ implică $O(\log n) = O(\ln n)$. Ca regulă avem ierarhia:

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \subset O(2^n)$$

Pentru o problemă dată, dorim să obținem un algoritm corespunzător unui ordin cât mai "la stânga". Pentru a simplifica calculele vom reduce din funcția polinomială, păstrând termenul cu gradul cel mai mare. De exemplu:

$$n^3 + 3n^2 + n + 8 \in O\big(n^3 + (3n^2 + n + 8)\big) = O(\max(n^3, 3n^2 + n + 8)) = O(n^3)$$

Deoarece notația se aplică pentru un n suficient de mare, relația de mai sus este adevărată. Chiar dacă ceilalți coeficienți ar fi fost negativi relația este adevărată:

$$n^3 - 3n^2 - n - 8 \in O\left(\frac{n^3}{2} + \left(\frac{n^3}{2} - 3n^2 - n - 8\right)\right)$$

$$= O\left(\max\left(\frac{n^3}{2}, \left(\frac{n^3}{2} - 3n^2 - n - 8\right)\right) = O\left(\frac{n^3}{2}\right) = O(n^3)\right)$$

chiar dacă pentru $0 \leq n \leq 6$ polinomul este negativ.

Notatia asimptotică conditionată

Mai mulți algoritmi sunt mai ușor de analizat dacă vom considera inițial cazuri a căror mărime satisface anumite condiții, de exemplu să fie puteri ale lui 2. În astfel de situații, folosim notația *asimptotică condiționată*. Fie $f : N \to R^*$ o funcție arbitrară și fie $P : N \to B$ un predicat.

$$O(f|P) = \left\{ t : N \to R^* \middle| \begin{array}{c} exist\check{a}\ o\ constant\check{a}\ pozitivp\ c\ \text{și}\ n_0\ astfel\ \hat{i}nc\hat{a}t \\ n \geq n_0\ \text{și}\ P(n) \Rightarrow t(n) \leq c \cdot f(n) \end{array} \right\}$$

Notația $O(f)$ este echivalentă cu $O(f|P)$ unde P este predicatul a cărui valoare este mereu *true*. Similar pentru notațiile $\Omega(f|P)$ și $\Theta(f|P)$.

O funcție $f : N \to R^*$ este eventual nedescrescătoare, dacă există un n_0 astfel încât pentru orice $n \geq n_0$ și orice $m \geq n$ avem $f(n) \leq f(m)$.

O funcție eventual *nedescrescătoare* este b-*netedă* dacă $f(bn) \in O(f(n))$. Orice funcție care este b-netedă pentru un anumit $b \geq 2$ este de asemenea b-netedă pentru orice $b \geq 2$.

Proprietatea 1

Fie $b \geq 2$ un întreg oarecare, $f: N \to R^*$ o funcție netedă și $t: N \to R^*$ o funcție eventual nedescrescătoare, astfel încât

$$t(n) = \begin{cases} a & pentru\ n = 1 \\ t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + bn\ pentru\ n \neq 1 \end{cases}$$

unde a, b sunt constante arbitrare reale pozitive. Este dificil să analizăm această ecuație. Considerăm doar cazurile când n este o putere a lui 2, atunci ecuația devine:

$$t(n) = \begin{cases} a & pentru\ n = 1 \\ 2t(n/2) + bn\ pentru\ n \neq 1 \end{cases}$$

Prin tehnicile prezentate mai jos vom ajunge la relația

$$t(n) \in \Theta(n \log n\ \ dac\check{a}\ n\ este\ o\ putere\ a\ lui\ 2)$$

Pentru a arăta că $t \in \Theta(\mathrm{n} \log \mathrm{n})$ trebuie să verificăm dacă t este eventual nedescrescătoare și $n \log n$ este netedă.

Prin inducție vom demonstra că $\forall n \geq 1\ avem\ \ t(n) \leq t(n+1)$. Pentru început notăm

$$t(1) = a \leq 2(a+b) = t(2)$$

Fie $n > 1$. Presupunem că pentru orice $m < n$ avem $t(m) \leq t(m+1)$. În particular,

$$t(\lfloor n/2 \rfloor) \leq t(\lfloor (n+1)/2 \rfloor)$$

$$t(\lceil n/2 \rceil) \leq t(\lfloor \lceil (n+1)/2 \rceil \rfloor)$$

Atunci, $t(n) = t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + bn \leq t(\lfloor (n+1)/2 \rfloor) + t(\lceil (n+1)/2 \rceil) + b(n+1) = t(n+1)$

Trebuie, de asemenea, să arătăm că $n \log n$ este netedă. Funcția este eventual nedescrescătoare și

$$2n \log 2n = 2n(\log 2 + \log n) = (2 \log 2\, n) + 2n \log n \in O(n + n \log n) = O(\max(n, n \log n))$$
$$= O(n \log n)$$

De multe ori, timpul de execuție al unui algoritm se exprimă sub forma unor inegalități de forma

$$t(n) \leq \begin{cases} t_1(n) & pentru\ \ n \leq n_0 \\ t(\lfloor n/2 \rfloor + \lceil n/2 \rceil) + cn & pentru\, n > n_0 \end{cases}$$

și

$$t(n) \geq \begin{cases} t_2(n) & pentru\ \ n \leq n_0 \\ t(\lfloor n/2 \rfloor + \lceil n/2 \rceil) + dn & pentru\, n > n_0 \end{cases}$$

Pentru anumite constante $c, d\ \in R^*$, $n_0 \in N$ și pentru funcții $t_1, t_2 \colon N \to R^+$.

Putem scrie cele două inegalități astfel:

$$t(n) \in t(\lfloor n/2 \rfloor + \lceil n/2 \rceil) + O(n)$$

$$t(n) \in t(\lfloor n/2 \rfloor + \lceil n/2 \rceil) + \Omega(n)$$

Aceste două expresii pot fi scrise și concentrat

$$t(n) \in t(\lfloor n/2 \rfloor + \lceil n/2 \rceil) + \Theta(n)$$

Definim funcția

$$f(n) = \begin{cases} 1 & pentru\, n = 1 \\ f(\lfloor n/2 \rfloor + \lceil n/2 \rceil) + n & pentru\, n \neq 1 \end{cases}$$

Am văzut că $f \in \Theta(n \log n)$. Ne întoarcem la funcția t . prin inducție se demonstrează că există constantele $v \leq d, u \geq c$ astfel ca

$$v \leq t(n)/f(n) \leq u$$

pentru orice $n \in N^+$. Deducem atunci că

$$t \in \Theta(f) = \Theta(n \log n)$$

Tehnici de analiză a algoritmilor

Nu există formulă generală pentru analiza eficienței unui algoritm. Aceasta este mai curând o chestiune de raționament, intuiție și experiență. Pe baza exemplelor de mai jos vom arata cum se poate efectua diverse analize.

Analiza algoritmului de însumare a elementelor din șir

Operație de bază : adăugarea unui element din șir la *sumă*.

Mărime de intrare: n, numărul de elemente din șir.

Indiferent de valorile numerelor din șir, vor fi n parcurgeri prin bucla *for*. De aceea $T(n) = n$

Analiza algoritmului de sortare prin interschimbare

Algoritmul sortează, comparând cheile, și schimbă dacă este cazul între două elemente, menținând ordinea în șir. Vom parcurge șirul în două for-uri imbricate după *i* și *j*.

Operație de bază: compară $S[j]$ cu $S[i]$.

Mărime de intrare: n, numărul de elemente de sortat.

Pentru un n dat, vor avea $n-1$ parcurgeri ale *for*-ului pe variabila *i*. La primul pas din *for*-ul pe *i*, sunt $n-1$ parcurgeri pe *for*-ul ce are contorul *j*. În al doilea pas există $n-2$ parcurgeri în *for*-ul *j*, ș.a.m.d. Numărul de parcurgeri ale *for*-ului *i* este:

$$T(n) = (n-1) + (n-2) + \cdots + 1 = \frac{(n-1)n}{2}$$

Analiza algoritmului de înmultirea a două matrice

În cazul acesta, există trei *for*-uri imbricate, iar cel mai interior, efectuează o multiplicare și o adunare.

Operația de bază: multiplicarea din cel mai interior *for*.

Mărimea de intrare: *n* numărul de coloane și rânduri.

Vor fi întotdeauna n pași efectuați în fiecare for (cel de variabilă *i*, cel de variabilă *j* și cel de variabilă *k*).

$$T(n) = n \times n \times n = n^3$$

Analiza algoritmului de căutare secvențială – cazul cel mai rău, cazul mediu

După cum am văzut, operația de bază nu este aceeași pentru toate instanțele de mărime n ale unui algoritm. Să considerăm numărul maxim de operații de bază pe care un algoritm le va face pentru a îndeplini sarcina. Acesta se notează cu $W(n)$ și se numește *complexitate a celui mai rău caz*. Dacă $T(n)$ există, evident că $W(n) = T(n)$. Cazul căutării secvențiale este unul elocvent, pentru că $T(n)$ nu există.

Operație de bază : comparația unui element din șir cu x.

Mărimea de intrare : n, numărul de elemente din șir.

Operația de bază este executată în cel mai rău caz de n ori, când x nu se află printre elementele din șir. De aceea $W(n) = n$.

Deși cel mai rău caz ne informează despre timpul maxim absolut consumat, în unele cazuri suntem interesați de timpul *mediu* necesar unui algoritm. Pentru un algoritm, fie $A(n)$ numărul de ori de câte ori algoritmul execută, în medie, operația de bază, pentru o intrare n. $A(n)$ se numește *complexitate medie*. La fel ca în cazul celui mai rău caz, dacă $T(n)$ există, $A(n) = T(n)$.

În cele ce urmează vom analiza cazul mediu în căutarea secvențială.

Operație de bază: comparația unui element din șir cu x.

Mărimea de intrare: n, numărul de elemente din șir.

Prima dată, vom analiza cazul în care x se află în S, toate elementele din S sunt distincte. Astfel, pentru un k, cu $1 \leq k \leq n$, probabilitatea ca x să se găsească pe poziția k este $1/n$. Dacă x se află pe poziția k, avem nevoie de k operații pentru a-l localiza pe x. Atunci timpul mediu este:

$$A(n) = \sum_{k=1}^{n} \left(k \times \frac{1}{n}\right) = \frac{1}{n} \times \sum_{k=1}^{n} k = \frac{1}{n} \times \frac{n(n+1)}{2} = \frac{n+1}{2}$$

Așa cum era de așteptat, în acest caz, aproximativ jumătate din șir este parcurs.

Vom analiza în cele ce urmează cazul în care x s-ar putea să nu fie în șir.

Atribuim o probabilitate p, evenimentului că x se află în șir. Dacă x se află în șir, atunci el se va afla pe una din pozițiile de la 1 la n. Probabilitatea ca x să se afle pe poziția k este $\frac{p}{n}$, iar probabilitatea ca să nu se afle în șir este $1 - p$. Astfel timpul mediu este:

$$A(n) = \sum_{k=1}^{n} \left(k \times \frac{p}{n}\right) + n(1-p) = \frac{p}{n} \times \frac{n(n+1)}{2} + n(1-p) = n\left(1 - \frac{p}{2}\right) + \frac{p}{2}$$

Dacă $p = 1$ atunci $A(n) = (n+1)/2$ și obțineam cazul anterior, în care suntem siguri că x se află printre elementele lui $A(n)$.

Sortarea prin selectie

Considerăm algoritmul de sortare prin selecție prezentat în primul curs. Timpul pentru o singură execuție a buclei interioare poate fi mărginit superior de o constantă a. În total, pentru un *i* dat, bucla interioară necesită un timp de cel mult $b + a(n-i)$ unități, unde b este constanta ce reprezintă timpul necesar inițializării buclei. O singură execuție a buclei exterioare are loc în cel mult $c + b + a(n-i)$ unități, unde c este o altă constantă. Algoritmul durează cel mult

$$d + \sum_{i=1}^{n-1} (c + b + a(n-i))$$

unități de timp, unde d este o altă constantă.

Simplificând, obținem

$$(a/2)n^2 + \left(b + c + \frac{a}{2}\right) n + (d - c - b)$$

De unde deducem că algoritmul necesită un ordin de timp $O(n^2)$. Nu este necesar să considerăm cazul mediu deoarece timpul este independent de o ordonare în prealabil a elementelor.

De obicei, detalii ca inițializarea buclei, nu se vor considera explicit. Pentru cele mai multe situații, este suficient să alegem ca *barometru* o anumită instrucțiune din algoritm și să numărăm de câte ori se execută această instrucțiune.

Sortarea prin insertie

Timpul pentru acest algoritm, prezentat tot în primul curs, este dependent de ordonarea prealabilă a elementelor de sortat. Folosim comparația $A[i] > key$ ca barometru. Să presupunem că j este fixat și fie $key = A[j]$. Cel mai nefavorabil caz apare când $x < A[i]$ pentru fiecare *i* între 2 și $i-1$. Numărul total de comparații pentru cazul cel mai nefavorabil este:

$$\sum_{i=1}^{n} (i-1) = \frac{n(-1)}{2} \in O(n^2)$$

Vom estima timpul mediu necesar pentru un caz oarecare. Presupunem că elementele tabloului A sunt distincte și că orice permutare a lor are aceeași probabilitate de apariție.

Dacă $1 \leq k \leq i$, probabilitatea ca $A[i]$ să fie al k-lea și cel mai mare dintre elementele $A[1] \dots A[i]$ este $1/i$. Pentru un i fixat condiția $A[i] < A[i-1]$ este falsă cu probabilitatea $1/i$. Probabilitatea ca să se execute comparația $x < A[i]$ o singură dată înainte de ieșirea din buclă este $1/i$. Probabilitatea ca și comparația se poate executa de exact două ori, este tot de $1/i$. Probabilitatea ca și comparația să se execute de exact $i-1$ ori este $2/i$ deoarece se întâmplă când $x < A[1]$ și când $A[1] \leq key < A[2]$. Pentru un număr fixat, numărul mediu de comparații este:

$$c_i = 1 \cdot \frac{1}{i} + 2 \cdot \frac{1}{i} + \cdots (i-2) \cdot \frac{1}{i} + \ (i-1) \cdot \frac{2}{i} = \frac{i+1}{2} - \frac{1}{i}$$

Pentru a sorta cele n elemente vom avea nevoie de $\sum_{j=2}^{n} c_j$ adică:

$$\frac{n^2+3n}{4} - H_n \in \Theta(n^2)$$

Unde prin $H_n = \sum_{i=1}^{n} i^{-1} \in \Theta(\log n)$ am notat al n-lea element al seriei armonice.

În cazul mediu, algoritmul efectuează de două ori mai puține comparații decât pentru cazul cel mai rău. totuși numărul comparațiilor este în ambele situații în $\Theta(n^2)$.

Heapsort

Vom analiza prima dată algoritmul *make-heap* din cursul anterior. Definim ca barometru instrucțiunile din bucla *repeat* din algoritmul de *sift-down*. Fie *m* numărul maxim de repetări al acestei bucle, cauzat de apelul lui $sift - down(T, i)$ unde i este fixat. Notăm cu j_t valoarea lui j după ce se execută instrucțiunea $j \leftarrow k$ la at-a repetare a buclei. Pentru t=1 $j_1 = i$. Dacă $1 < t \leq m$, la sfârșitul celei de-a $(t-1)$-a repetări a buclei avem $j \neq k$ și $k \geq 2j$. În general $j_t \geq 2j_{t-1}$ pentru $1 < t \leq m$.

Atunci,

$$n \geq j_m \geq 2j_{m-1} \geq 4j_{m-2} \geq \cdots \geq 2^{m-1}j$$

Rezultă că $2^{m-1} \leq \frac{n}{i}$ iar de aici rezultă relația$m \leq 1 + \log\left(\frac{n}{i}\right)$.

Numărul total de executări ale buclei **repeat** la formarea unui heap este mărginit superior de

$\sum_{i=1}^{a}\left(1 + \log\frac{n}{i}\right)$ unde $a = \lfloor n/2 \rfloor$

Pentru a simplifica această expresie, să observăm că pentru orice $k \geq 0$

$\sum_{i=b}^{c} \log\left(\frac{n}{i}\right) \leq 2^k \log\left(\frac{n}{2^k}\right)$ unde $b = 2^k$ şi $c = 2^{k+1} - 1$

Descompunem expresia de mai sus în secțiuni ale puterilor lui 2 și notăm $d = \lfloor \log(n/2) \rfloor$

$$\sum_{i=b}^{c} \log\left(\frac{n}{i}\right) \leq \sum_{k=0}^{d} 2^k \log\frac{n}{2^k} \leq 2^{d+1} \log\frac{n}{2^{d-1}}$$

Această inegalitate pornește de la egalitatea:

$$\sum_{k=0}^{d} 2^k \log\left(\frac{n}{2^k}\right) = 2^{d+1} \log\left(\frac{n}{2^{d-1}}\right) - 2 - \log n$$

Egalitatea de mai sus poate fi demonstrată la rândul ei prim formula:

$$\sum_{k=0}^{d} 2^k \log\left({}^{n}/_{2^k}\right) = \left(2^{d+1} - 1\right) \log n - \sum_{k=0}^{d} (2^k k)$$

Mai rămâne să demonstrăm că

$$\sum_{k=0}^{d} \left(2^k k\right) = (d-1)2^{d+1} + 2$$

Dar $d = \left\lfloor \log\left(\frac{n}{2}\right) \right\rfloor$ implică faptul că $d + 1 \leq \log n$ și $d - 1 \geq \log \frac{n}{8}$. În concluzie

$$\sum_{i=1}^{a} \log\left(\frac{n}{i}\right) \leq 3n$$

Deducem astfel că *make-heap* necesită un timp de $t \in O(n)$. Pe de altă parte, deoarece orice algoritm pentru formarea unui heap trebuie să utilizeze fiecare element cel puțin odată rezultă că $t \in \Omega(n)$. Cu alte cuvinte, $t \in \Theta(\mathrm{n})$. Se poate compara acest algoritm cu timpul de rulare al *slow-make-heap*.

Pentru cel mai nefavorabil caz, $sift\_down(T[1..n-1], 1)$ necesită un timp în $O(\log n)$. Ținând cont că make-heap este liniar, rezultă că algoritmul *heapsort* are un ordin de timp $O(n \log n)$. Acest lucru este valabil atât pentru cazul nefavorabil cât și pentru cazul mediu.

Turnurile din Hanoi

Această problemă a fost descrisă în primul curs.

Observăm că pentru a muta cele mai mici n discuri de pe tija i pe tija j (unde $1 \leq i \leq 3, 1 \leq j \leq 3, i \neq j, n \geq 1$), transferăm cele mai mici n-1 discuri de pe tija i pe tija $6 - i - j$, apoi transferăm discul

n de pe tija i pe tija j, apoi retransferăm cele n-1 discuri de pe tija $6 - i - j$ pe tija j. Astfel reducem problema mutării a n discuri la problema mutării a $n - 1$ discuri.

Următoarea procedură descrie acest algoritm.

$HANOI(n, i, j)$

1. $\triangleright$ *mută cele mai mici n discuri de pe tija i pe tija j*
2. ***if*** $n > 0$ ***then***
3. $\quad HANOI(n - 1, i, 6 - i - j)$
4. $\quad$ ***scrie*** $i \rightarrow j$
5. $\quad HANOI(n - 1, 6 - i - j, j)$

Apelul inițial este $HANOI(64,1,2)$

Considerăm instrucțiunea *scrie* ca barometru. Timpul necesar algoritmului este dat de următoarea recurență:

$$t(n) = \begin{cases} 1 & pentru\ n = 1 \\ 2t(n-1) + 1 & pentru\ n > 1 \end{cases}$$

Rezultă că $t(n) = 2^n - 1$. Vom demonstra acest lucru în cele ce urmează.

Analiza algoritmilor recursivi

Cel mai important câștig al recursivității este faptul că ea este naturală, și compactă, fără să ascundă esența algoritmului prin detaliile din implementare. Pe de altă parte, apelurile recursive trebuie folosite cu discernământ, deoarece solicită resursele calculatorului. Iată câteva tehnici de analizare a algoritmilor recursivi.

Metoda iteratiei

Se execută primii pași, se intuiește forma generală, iar apoi se demonstrează prin inducție matematică că forma este corectă. Să considerăm de exemplu recurența problemei turnurilor din Hanoi. Pentru un anumit $n > 1$ obținem succesiv

$t(n) = 2t(n-1) + 2^0 = 2^2 t(n-2) + 2^0 + 2^1 = \cdots = 2^{n-1} t(1) + \sum_{i=0}^{n-2} 2^i$

Rezultă $t(n) = 2^n - 1$

Inductia constructivă

Inducția matematică este folosită de obicei ca tehnică de demonstrare a unei aserțiuni deja enunțate. Vom vedea în această secțiune că inducția matematică poate fi utilizată cu succes și în descoperirea enunțului aserțiunii. Aplicând această tehnică, putem simultan să demonstrăm o aserțiune doar parțial specificată și să descoperim specificațiile care lipsesc și datorită cărora aserțiunea este corectă. Vom vedea că această tehnică este utilă pentru rezolvarea unor recurențe ce apar în contextul analizei algoritmilor.

Fie funcția $f: N \to N$, definită prin recurența

$$f(n) = \begin{cases} 0 & pentru\ n = 0 \\ f(n-1) + n & pentru\ n > 0 \end{cases}$$

Să presupunem că nu știm că $f(n) = \frac{n(n+1)}{2}$ și să căutăm această formulă.

Avem $f(n) = \sum_{i=0}^{n} i \leq \sum_{i=0}^{n} n = n^2$

Deci $f(n) \in O(n^2)$. Aceasta ne sugerează ipoteza inducției conform căreia $f(n) = an^2 + bn + c$. Această ipoteză este parțială, adică a, b, c, nu sunt cunoscute deocamdată. Tehnica inducției constructive constă în a demonstra prin inducție matematică această ipoteză incompletă și a determina în același timp valorile constantelor necunoscute. Presupunem ipoteza de mai sus valabilă pentru $n \geq 1$:

$$f(n) = a(n-1)^2 + b(n-1) + c + n = an^2 + (1 + b - 2a)n + (a - b + c)$$

Dacă dorim să arătăm ca formula este valabilă pentru n, trebuie să arătăm că $f(n) = an^2 + bn + c$. Prin identificarea coeficienților puterilor lui *n* obținem ecuațiile

$$1 + b - 2a = b$$

$$a - b + c = c$$

Cu soluția $a = b = 1/2$ iar c putând fi orice număr.

Rezultă că $f(n) = \frac{n^2}{2} + \frac{n}{2} + c$. Rămâne să arătăm că relația este valabilă și pentru primul pas, 0.

$f(0) = a \cdot 0 + b \cdot 0 + c = c$. Dar cum $f(n) = 0$ pentru $n = 0$ avem $c = 0$.

Rezultă că $f(n) = \frac{n^2}{2} + \frac{n}{2}$ pentru orice n.

Recurente liniare omogene

Există și tehnici ce pot fi folosite aproape automat pentru a rezolva anumite clase de recurențe. Vom începe prin a considera ecuații recurente liniare omogene, adică de forma

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = 0$$

unde t_i sunt valorile căutate iar coeficienții a_i sunt constante.

Conform intuiției vom căuta soluții de forma

$$t_n = x^n$$

Încercăm această soluție și obținem

$$a_0 x^n + a_1 x^{n-1} + \cdots + a_k x^{n-k} = 0$$

Soluția trivială nu ne interesează. Presupunând că avem k rădăcini ale ecuației rezultă:

$$t_n = \sum_{i=1}^{k} c_i r_i^n$$

Orice combinație este soluție a recurenței, unde constantele $c_1, c_2, \ldots, c_k$ sunt determinate de condițiile inițiale. Vom exemplifica prin recurența care definește șirul Fibonacci:

$$t_n = t_{n-1} - t_{n-2} \quad n \geq 2$$

Iar $t_0 = 0, t_1 = 1$. Putem să rescriem această recurența ca

$$t_n - t_{n-1} - t_{n-2} = 0$$

Ecuația caracteristică este:

$$x^2 - x - 1 = 0$$

Rădăcinile acestei ecuații sunt $r_{1,2} = \frac{1 \pm \sqrt{5}}{2}$. Soluția generală are forma:

$$t_n = c_1 r_1^n + c_2 r_2^n$$

Impunând condițiile inițiale obținem

$$c_1 + c_2 = 0 \qquad n = 0$$

$$r_1 c_1 + r_2 c_2 = 1 \quad n = 1$$

de unde determinăm $c_{1,2} = \pm \frac{1}{\sqrt{5}}$

$$t_n = \frac{1}{\sqrt{5}} (r_1^n - r_2^n)$$

$r_1 = \theta = \frac{1+\sqrt{5}}{2}$ iar $r_2 = -\theta^{-1}$

Obținem $t_n = \frac{1}{\sqrt{5}}(\theta^n - (\theta)^{-n})$

Se poate vedea că algoritmul lui Fibonacci are un ordin exponențial.

Ce facem dacă rădăcinile ecuației caracteristice nu sunt distincte? Se poate arăta că dacă *r* este o rădăcină de multiplicitate *m*, a ecuației caracteristice atunci $t_n = r^n, t_n = nr^n, t_n = n^2r^n, \dots, t_n = n^{m-1}r^n$.

Soluția generală pentru o astfel de recurență este o combinație liniară a acestpr termeni și a termenilor proveniți din celelalte rădăcini ale ecuației caracteristice.

Recurente liniare neomogene

Considerăm următoarea formulă generală pentru recurențe:

$$a_0t_n + a_1t_{n-1} + \cdots + a_kt_{n-k} = b^np(n)$$

unde b este o constantă, iar $p(n)$ este un polinom in n de grad d. Ideea este că prin manipulări convenabile, putem reduce un astfel de caz la o formă omogenă.

Un exemplu pentru o astfel de recurență poate fi :

$$t_n - 2t_{n-1} = 3^n$$

În acest caz $b = 3, p(n) = 1$. Înmulțim recurența cu 3 rezultă

$$3t_n - 6t_{n-1} = 3^{n+1}$$

Pentru $n + 1$ avem

$$3t_{n+1} - 6t_n = 3^{n+1}$$

Dacă scădem cele două ecuații obținem

$$t_{n+1} - 5t_n + 6t_{n-1} = 0$$

Am obținut o ecuație de recurență omogenă, pe care o putem rezolva ca pe precedenta. Ecuația caracteristică este:

$$x^2 - 5x + 6 = 0$$

adică $(x - 2)(x - 3) = 0$. Se observă că factorul $(x - 2)$ corespunde părții stângi a recurenței inițiale iar $(x - 3)$ a apărut ca rezultat al manipulărilor efectuate, pentru a elimina partea dreapta.

Generalizând, să luăm ecuația caracteristică:

$$\left(a_0x^k + a_1x^{k-1} + \cdots + a_k\right)(x - b)^{d+1} = 0$$

Vom rezolva acum recurența corespunzătoare problemei turnurilor din Hanoi:

$$t_n = 2t_{n-1} + 1 \quad n \geq 1$$

Iar $t_0 = 0$. Rescriem recurența astfel: $t_n - 2t_{n-1} = 1$

care este de forma de mai sus cu $b = 1, p(n) = 1$. Ecuația caracteristică este $(x-2)(x-1) = 0$ cu soluțiile 1,2. Soluția generală este $t_n = c_1 1^n + c_2 2^n$. Avem nevoie de două condiții inițiale. Una este $t_0 = 0$. Pentru a doua calculăm $t_1 = 2t_0 + 1$. Rezultă

$$t_n = 2^n - 1$$

Rezultă astfel că ordinul lui t_n este $O(2^n)$.

Schimbarea variabilei

Uneori, printr-o schimbare de variabilă, putem rezolva recurențe mult mai complicate.

Iată un exemplu în acest sens:

$$T(n) = 4T\left(\frac{n}{2}\right) + n \quad n > 1$$

în care înlocuim pe n cu 2^k, notăm cu $t_k = T(2^k) = T(n)$. Obținem

$$t_k = 4t_{k-1} + 2^k$$

Ecuația caracteristică a acestei recurențe este:

$$(x-4)(x-2) = 0$$

și deci, $t_k = c_1 4^k + c_2 2^k$. Înlocuim pe k cu $\log n$ rezultă:

$$T(n) = c_1 n^2 + c_2 n$$

Rezultă că $T(n) \in O(n^2)$ unde n este o putere a lui 2.