ANALIZA ALGORITMILOR

Acest capitol vă va familiariza cu concepţia de bază folosite în această carte, referitor la proiectarea şi analiza algoritmilor.

Începem cu o discuţie asupra problemelor generale ale calculabilităţii şi ale algoritmilor necesari pentru rezolvarea acestora, cu problema sortării, ca exemplu introductiv. Pentru a arăta cum vom specifica algoritmii prezentaţi, vom introduce un "pseudocod", care ar trebui să fie familiar cititorilor obişnuiţi cu programarea. Sortarea prin inserţie, un algoritm simplu de sortare, va servi ca exemplu iniţial. Vom analiza timpul de execuţie pentru sortarea prin inserţie, introducând o notaţie care să descrie modul în care creşte acest timp odată cu numărul obiectelor aflate în operaţia de sortare. De asemenea vom introduce în proiectarea algoritmilor metoda *divide* şi *stăpâneşte,* pe care o vom utiliza pentru dezvoltarea unui algoritm numit *sortare prin interclasare.* Vom încheia cu o comparaţie între cei doi algoritmi de sortare.

1. Algoritmi

Fără a fi foarte exacţi, spunem că un ***algoritm*** este o procedură de calcul bine definită care primeşte o valoare sau o mulţime de valori ca ***date de intrare*** şi produce o anumită valoare sau o mulţime de valori ca ***date de ieşire***. Astfel ,un algoritm este un şir de paşi care transformă datele de intrare în date de ieşire.

Putem de asemenea să privim un algoritm ca pe un instrument de rezolvare a ***problemelor de calcul bine*** definite. Enunţul problemei specifică, în termeni generali, relaţia dorită intrare/ieşire. Algoritmul descrie o anumită procedură de calcul pentru a se ajunge la această legătură intrare/ieşire.

Vom începe studiul algoritmilor cu problema sortării unui şir de numere în ordine nedescrescătoare. Această problemă apare frecvent în practică şi furnizează o bază foarte utilă pentru introducerea multor metode standard pentru proiectarea şi analiza algoritmilor. Iată cum vom defini formal ***problema sortării***:

Date de intrare: Un şir de n numere $\langle a_1,a_2,\ldots,a_n\rangle$.

Date de ieşire: O permutare (reordonare) a şirului iniţial,$\langle a_1',a_2',\ldots,a_n'\rangle$ astfel încât $a_1' \le a_2' \le \ldots \le a_n'$.

Fiind dat un şir de intrare ca ,de exemplu$\langle 26,31,41,41,58,59\rangle$. Un şir de intrare ca cel de mai sus se numeşte ***o instanţă*** a problemei de sortare. În general o ***instanţă a unei probleme*** se va înţelege mulţimea tuturor datelor de intrare(care satisfac restricţiile impuse în definirea problemei)necesare pentru a determina o soluţie a problemei.

Sortarea este o soluţie fundamentală în informatică (multe programe o folosesc ca pas intermediar) şi, ca urmare, a fost dezvoltat un număr mare de algoritmi de sortare. Care algoritm este cel mai bun pentru o aplicaţie dată depinde de obiecte care trebuie sortate, de gradul în care aceste obiecte sunt deja sortate într-un anumit fel şi de tipul de mediu electronic care urmează să fie folosit: memorie principală, discuri sau benzi magnetice.

Un algoritm este ***corect*** dacă ,pentru orice instanţă a sa, se termină furnizând ieşirea corectă. Vom spune că un algoritm corect *rezolvă* problema de calcul dată. Un algoritm incorect s-ar putea să nu se termine deloc în cazul unor anumite instanţe de intrare, sau s-ar putea termina producând un alt răspuns decât cel dorit. Contrar a ceea ce s-ar putea crede, algoritmii incorecţi pot fi uneori utili dacă rata lor de eroare poate fi controlată.

Concret un algoritm poate fi specificat printr-un program pentru calculator sau chiar un echipament hardware. Singura condiţie este aceea ca specificaţiile să producă o descriere precisă a procedurii de calcul care urmează a fi parcursă.

La acest curs vom descrie algoritmii sub forma unor programe scrise într-un ***pseudocod*** care seamănă foarte mult cu limbajele C, Pascal sau Algol. Dacă sunteţi cât de cât familiarizaţi cu oricare din acestea, nu veţi avea probleme în a citi algoritmii noştri. Ceea ce diferenţiază codul real de pseudocod este faptul că vom folosi metoda cea mai clară şi mai concisă pentru a descrie un algoritm dat. O altă diferenţă dintre pseudocod şi codul real este acea că pseudocodul nu se ocupă de detalii de utilizare. Problemele abstractizării datelor sau a tratării erorilor sunt deseori ignorate, pentru a transmite cât mai concis esenţa algoritmului.

1.1 Sortarea prin inserţie

Începem cu sortarea prin inserţie, care este un algoritm eficient pentru sortarea unui număr mic de obiecte. Sortarea prin inserţie funcţionează în acelaşi fel în care mulţi oameni sortează un pachet de cărţi de joc obişnuite. Se începe cu pachetul aşezat pe masă cu faţa în jos şi cu mâna stângă goală. Apoi luăm câte o

carte de pe masă îi o inserăm în poziţia corectă în mâna stângă. Pentru a găsi poziţia corectă pentru o carte dată, o comparăm cu fiecare dintre cărţile aflate deja în mâna stângă, de la dreapta la stânga.

Pseudocodul pentru sortarea prin inserţie este prezentat ca o procedură numită **Sortează Prin - Inserţie** ,care are ca parametru un vector $A[1..n]$ conţinând un şir de lungime n, care urmează a fi sortat. (Pe parcursul codului, numărul de elemente ale lui A va fi notat cu $lungime[A]$.) Numerele de intrare sunt sortate pe loc, în cadrul aceluiaşi vector A; cel mult un număr constant dintre acestea sunt memorate în zone de memorie suplimentară. Când **Sortează Prin - Inserţie** se termină, vectorul iniţial A va conţine elementele şirului de ieşire sortat.

Sortează Prin - Inserţie(*A*)

```
1: pentru j←2, lungime[A] execută
2: cheie ← A[j]
3:▷Inserează A[j] în şirul sortat A[1..j-1]
4: I ← j -1
5: cât timp i>0 şi  cheie <  A[i] execută
6: A[i+1] ←A[i]
7:  i←i-1
8:  A[i+1] ← cheie
```

5	2	4	6	1	3	
2	5	4	6	1	3	
2	4	5	6	1	3	
2	4	5	6	1	3	
1	2	4	5	6	3	
1	2	3	4	5	6	stop

Figura 1.2.Modul de operare a procedurii **Sortează Prin - Inserţie** , asupra vectorului $A = \langle 5,2,4,6,1,3 \rangle$. Poziţia indicelui j este indicată printr-un cerc.

Figura 1.2 ilustrează modul de funcţionare a acestui algoritm pentru A = ‹ 5, 2, 4, 6, 1, 3›. Indicele j corespunde cărţii care urmează a fi inserată în mâna stângă. Elementele $A[1..j-1]$ corespund mulţimii de cărţi din mână, deja sortate, iar elementele $A[i+1..n]$ corespund pachetului de cărţi aflate pe masă. Indicele se deplasează de la stânga spre dreapta în interiorul vectorului. La fiecare iteraţie. elementul $A[j]$ este ales din vector (linia 2). Apoi, plecând de la poziţia j-1, elementele sunt, succesiv deplasate o poziţie spre dreapta până când este găsită poziţia corectă pentru $A[j]$ (liniile 4-7),moment în care acesta este înserat (linia 8).

Convenţii pentru pseudopod

La scrierea pseudocodului vom folosi următoarele convenţii:

1. Indentarea (spaţiu liber) indică o structură de bloc. De exemplu, corpul ciclului **pentru (for)**, care începe în linia 1, constă din liniile 2-8, iar corpul ciclului **cât timp,** care începe în linia 5, constă din liniile 6-7, dar nu şi linia 8. Stilul nostru de indentare se aplică şi structurilor de tipul **dacă – atunci - altfel.** Folosirea indentării în locul unor indicatori de bloc de tipul **begin** şi **end**, reduce cu mult riscul de confuzie, îmbunătăţind claritatea prezentării.
2. Ciclurile de tipul **cât timp, pentru, repetă** şi construcţiile condiţionale **dacă, atunci** şi **altfel** au aceiaşi interpretare ca şi structurile similare din Pascal**.**
3. Simbolul ▷ indică faptul că restul liniei este un comentariu.
4. O atribuire multiplă de forma $i \leftarrow j \leftarrow e$ înseamnă atribuirea valorii expresiei e ambelor variabile i şi j; aceasta ar trebui tratată ca un echivalent al atribuirii $i \leftarrow e$, urmată de atribuirea $i \leftarrow j$.

5. Variabilele (de exemplu *i, j, cheie*) sunt locale pentru o procedură dată. Nu vom utiliza variabile globale fără a preciza acest tip de lucru în mod explicit.
6. Elementele unui vector sunt accesate specificând numele vectorului urmat de indice în paranteze drepte. De exemplu $A[i]$ indică elementul de rang i al vectorului A. Notaţia ".." este folosită pentru a indica un domeniu de valori în cadrul unui vector. Astfel $A[1..j]$ indică un subvector al lui A constând din elementele $A[1], A[2] .. A[j]$.
7. Datele compuse sunt, în mod uzual, organizate în *obiecte* care conţin *atribute* sau *câmpuri.* Un anumit câmp este accesat folosind numele câmpului urmat de numele obiectului său în paranteze drepte. De exemplu, tratăm un vector ca pe un obiect cu atributul *lungime* indicând numărul de elemente ale acestuia. Pentru a specifica numărul de elemente ale unui vector *A,* se va scrie *lungime*[*A*]. Deşi vom folosi paranteze drepte atât pentru indexarea elementelor unui vector, cât şi pentru atributele obiectelor, va fi clar din context care este interpretarea corectă. O variabilă reprezentând un vector sau un obiect este tratată ca un pointer spre datele care reprezintă vectorul sau obiectul. Pentru toate câmpurile *f* ale unui obiect *x* atribuirea $y \leftarrow x$, are ca efect $f[y]=f[x]$. Mai mult dacă acum avem $f[x] \leftarrow 3$,atunci nu numai $f[y]=3$,dar şi $f[x]=3$. Cu alte cuvinte, x şi y indică spre acelaşi obiect după atribuirea $y \leftarrow x$. Uneori un pointer nu se va referi nici la un obiect. În acest caz special pointerul va primi valoarea NIL.
8. Parametrii sunt transmişi unei proceduri *prin valoare*: procedura apelată primeşte propria sa copie a parametrilor şi dacă atribuie o valoare unui parametru, schimbarea nu este văzută de procedura apelantă. Când obiectele sunt transmise procedurii, este copiat doar pointerul spre datele reprezentând obiectul, nu şi câmpurile acestuia. De exemplu dacă x este un parametru al unei proceduri apelate, atribuirea $y \leftarrow x$ în cadrul procedurii apelate nu este vizibilă din procedura apelantă. Atribuirea $f[x] \leftarrow 3$ este, totuşi, vizibilă.

2. Analiza algoritmilor

Analiza unui algoritm a ajuns să însemne prevederea resurselor pe care algoritmul le solicită. Uneori, resurse ca memoria, lăţimea benzii de comunicaţie, porţi logice sunt prima preocupare, dar de cele mai multe ori, vrem să măsurăm timpul de execuţie necesar algoritmului. În general, analizând mai mulţi algoritmi pentru o anumită problemă, cel mai eficient poate fi identificat uşor. O astfel de analiză poate indica mai mulţi candidaţi viabili, dar câţiva algoritmi inferiori sunt, de obicei, eliminaţi în timpul analizei.

Înainte de a analiza un algoritm, trebuie să avem un model al tehnologiilor de implementare care urmează să fie folosită, incluzând un model pentru resursele acesteia şi costurile corespunzătoare. În cea mai mare partea a acestei cărţi vom presupune că tehnologia de implementare este un model de calcul generic cu un procesor, *maşină cu acces aleator (RAM)* şi vom presupune că algoritmii vor fi implementaţi ca programe pentru calculator. În modelul RAM, instrucţiunile sunt executate una după alta, fără operaţii concurente.

Analiza, chiar şi a unui singur algoritm, poate fi, uneori, o încercare dificilă. Matematica necesară poate să includă combinatorică, teoria probabilităţilor, dexteritate algebrică şi abilitatea de a identifica cei mai importanţi termeni într-o expresie. Deoarece comportarea unui algoritm poate fi diferită în funcţie de datele de intrare, avem nevoie de un mijloc de exprimare a acestei comportări în formule simple, uşor înţelese. Deşi pentru a analiza un algoritm, selectăm nu numai un anumit tip de model de maşină de calcul, rămân mai multe posibilităţi de alegere a felului în care decidem să exprimăm această analiză. Un scop imediat este de a găsi un mijloc de exprimare care să fie simplu de scris şi de manevrat, care să arate principalele resurse necesare unui algoritm şi să suprime detaliile inutile.

2.1. Analiza sortării prin inserţie.

Timpul de execuţie necesar procedurii **Sortează – Prin - Inserţie** depinde de intrare: sortarea a o mie de numere ia mai mult timp ca sortarea a trei numere. Mai mult decât atât, **Sortează – Prin - Inserţie** poate să consume timpi diferiţi pentru a sorta două şiruri de numere de aceeaşi dimensiune, în funcţie de măsura în care acestea conţin numere aproape sortate. În general, timpul necesar unui algoritm creşte odată cu dimensiunea datelor de intrare, astfel încât este tradiţional să se descrie timpul de execuţie al unui program în

funcţie de dimensiunile datelor de intrare. În cest scop, trebuie să definim cu mai multă precizie termenii de "timp de execuţie" şi "dimensiunea datelor de intrare"

Definiţia *dimensiunea datelor de intrare* depinde de problema studiată. Pentru multe probleme, cum ar fi sortarea sau calculul unei transformate Fourier discrete, cea mai naturală măsură este *numărul de obiecte din datele de intrare*-de exemplu, pentru sortare, un vector de dimensiune *n*. Pentru multe alte probleme, ca înmulţirea a doi întregi, cea mai bună măsură pentru dimensiunea datelor de intrare este *numărul total de biţi* necesari pentru reprezentarea datelor de intrare în notaţie binară. Uneori este mai potrivit să exprimăm dimensiunea datelor de intrare prin două numere în loc de unul. De exemplu, dacă datele de intrare ale unui algoritm sunt reprezentate de un graf, datelor de intrare poate fi descrisă prin numărul de vârfuri şi muchii ale grafului. Pentru fiecare problemă pe care o vom studia, vom indica măsura utilizată pentru dimensiunea datelor de intrare.

Timp de execuţie a unui algoritm pentru un set de date de intrare este determinat de operaţii primitive sau "paşi" executaţi. Este util să definim noţiunea de "paşi", astfel încât să fie cât mai independent de calculator. Pentru moment, să adoptăm următorul punct de vedere. Pentru execuţia unei linii din pseudocod este necesară o durată constantă de timp. O anumită linie poate avea nevoie de un timp de execuţie decât o alta, dar vom presupune că fiecare execuţie a liniei *i* consumă timpul c_i, unde c_1 este o constantă. Acest punct de vedere este conform cu modelul RAM şi în acelaşi timp, reflectă, destul de bine, modul în care pseudocodul poate fi utilizat în cele mai multe cazuri concrete.

În prezentarea care urmează, expresia noastră pentru timpul de execuţie al algoritmului **Sortează – Prin - Inserţie** va evolua de la o formulă relativ complicată, care foloseşte toate costurile de timp c_i, la una

Mult mai simplă din notaţii, care este mai concisă şi mai uşor de manevrat. Această notaţie mai simplă va face, de asemenea, uşor determinat dacă un algoritm este mai eficient decât altul.

Începem prin a relua prezentarea procedurii **Sortează – Prin - Inserţie**, adăugând costul de timp pentru fiecare instrucţiune şi un număr care reprezintă de câte ori aceasta este efectiv execută. Pentru fiecare j=2,3,…,n; $n = lungime[A]$, vom nota cu t_j numărul de execuţii ale testului cât timp din linia 5 pentru valoarea fixată j. Vom presupune că un comentariu nu este o instrucţiune executabilă, prin urmare nu cere timp de calcul. Timpul de execuţie al algoritmului este suma tuturor timpilor de execuţie corespunzători

Sortează – Prin - Inserţie(A)	***cost***	***timp***
1: **pentru** $j \leftarrow 2$, $lungime[A]$ **execută**	c_1	n
2: $cheie \leftarrow A[j]$	c_2	n-1
3:▷Inserează $A[j]$ în şirul sortat $A[1..j\text{-}1]$	0	n-1
4: $i \leftarrow$ j -	c_4	n-1
5: **cât timp** $i>0$ şi $cheie < A[i]$ **execută**	c_5	$\sum_{j=2}^{n} t_j$
6: $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7: $i \leftarrow i$-1	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8: $A[i+1] \leftarrow cheie$	c_8	n-1

fiecărei instrucţiuni executate: o instrucţiune care consumă timpul c_i pentru execuţie şi este executată de n ori ,va contribui cu $c_i n$ la timpul total de execuţie. Pentru a calcula $T(n)$ timpul de execuţie pentru **Sortează – Prin - Inserţie,** vom aduna produsele mărimilor indicate în coloanele *cost* şi *timp*, obţinând:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1).$$

Chiar pentru datele de intrare de aceeaşi marime, timpul de execuţie al unui algoritm dat poate să depindă de conţinutul datelor de intrare. De exemplu, pentru **Sortează – Prin - Inserţie**, cazul cel mai favorabil apare când vectorul de intrare este deja sortat. Pentru fiecare j = 2, 3, .. n, vom găsi că $A[i] \le cheie$ în linia 5, când i are valoarea iniţială j-1.Rezultă t_j = 1 pentru j = 2, 3, .. n şi timpul de execuţie în cazul cel mai defavorabil este:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) =$$
$$. = (c_1 + c_2 + c_4 + c_8)n - (c_5 + c_2 + c_4 + c_8).$$

Acest timp de execuţie poate fi exprimat sub forma *an+ b* pentru anumite *constante a* şi *b* care depind de timpii de execuţie c_i fiind astfel o *funcţie lineară* de *n.*

Dacă vectorul este sortat în ordine inversă – adică în ordine descrescătoare-obţinem cazul cel mai defavorabil. În această situaţie trebuie să comparăm fiecare element *A*[*j*] cu fiecare element din subvectorul *A*[*1..j-1*], şi astfel $t_j = j$ pentru $j = 2, 3, .. n$. Observând că

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

şi

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2} - 1$$

găsim că în cazul cel mai defavorabil timpul de execuţie pentru **Sortează – Prin - Inserţie** este

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$
$$. = \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} + c_8\right)n - (c_5 + c_2 + c_4 + c_8).$$

Rezultă că timpul de execuţie în cazul cel mai defavorabil poate fi exprimat sub forma $an^2 + bn + c$, unde *a, b, c* depind, din nou de costurile c_i ale instrucţiunilor, fiind astfel o *funcţie pătratică* de *n* .

De obicei, la fel ca la sortarea prin inserţie, timpul de execuţie al unui algoritm dat este fix pentru anumite date de intrare. Totuşi, în ultimele capitole, vom întâlni câţiva algoritmi "aleatori" a căror comportare poate varia chiar pentru aceleaşi date de intrare.

2.2. Analiza celui mai defavorabil caz şi a cazului mediu

În analiza sortării prin inserţie am cercetat ambele situaţii extreme: cazul cel mai favorabil, în care vectorul de intrare era deja sortat, respectiv, cel mai defavorabil ,în care vectorul de intrare era sortat în ordine inversă. În continuare, ne vom concentra, de regulă pe găsirea *timpului de execuţie în cazul cel mai defavorabil*, cu alte cuvinte ,a celui mai mare timp de execuţie posibil relativ la orice date de intrare de dimensiune constantă *n*. Precizăm trei motive pentru această orientare:

- Timpul de execuţie al unui algoritm în cazul cel mai defavorabil este o margine superioară a timpului de execuţie pentru orice date de intrare de dimensiune fixă. Cunoscând acest timp avem o garanţie că timpul nu va avea, niciodată, un timp de execuţie mai mare. Nu va fi nevoie să facem presupuneri sau investigaţii suplimentare asupra timpului de execuţie şi să sperăm că acesta nu va fi, niciodată mult mai mare.
- Pentru anumiţi algoritmi, cazul cel mai defavorabil apare destul de frecvent. De exemplu, în căutarea unei anumite informaţii într-o bază de date, cazul cel mai defavorabil al algoritmului de căutare va apare deseori când informaţia căutată nu este ,de fapt, prezentă în baza de date. În anumite aplicaţii, căutarea unor informaţii absente poate fi frecventă.
- "Cazul mediu "este, adesea, aproape la fel de defavorabil ca şi cazul cel mai defavorabil. Să presupunem că alegem la întâmplare *n* numere şi aplicăm sortarea prin inserţie. Cât timp va fi necesar pentru a determina locul în care putem însera *A*[*j*] în subvectorul *A*[1..*j-1*] sunt mai mici ca *A*[*j*], şi cealaltă jumătate sunt mai mari. Prin urmare, în medie, trebuie verificate jumătate din

elementele subvectorului $A[1..j-1]$, deci $t_j=j/2$. Dacă ţinem seama de această observaţie, timpul de execuţie mediu va apărea tot ca o funcţie pătratică de n , la fel ca în cazul cel mai defavorabil.

În anumite cazuri particulare, vom fi interesaţi de *timpul mediu de execuţie* al unui algoritm. O problemă care apare în analiza cazului mediu este aceea că s-ar putea să nu fie prea clar din ce sunt constituite datele de intrare "medii" pentru o anumită problemă. Adesea vom presupune că toate datele de intrare având o dimensiune dată sunt la fel de probabile. În practică această presupunere poate fi falsă,dar un algoritm aleator poate, uneori, să o forţeze.

2.3. Ordinul de creştere

Pentru a uşura analiza procedurii **Sortează – Prin - Inserţie,** am utilizat mai multe presupuneri simplificatoare. În primul rînd, am ignorat costul real al fiecărei instrucţiuni, folosind constantele c_i pentru a reprezenta aceste costuri. Apoi am observat că prin aceste constante obţinem mai multe detalii decât avem nevoie în mod real: timpul de execuţie în cazul cel mai defavorabil este de forma an^2+bn+c pentru anumite constante a, b, c care depind de costurile c_i ale instrucţiunilor. Astfel am ignorat nu numai costurile reale ale instrucţiunilor, dar şi costurile abstracte c_i .

Vom face acum încă o abstractizare simplificatoare. Ceea ce ne interesează de fapt, este ***rata de creştere*** sau ***ordinul de creştere*** a timpului de execuţie. Considerăm, prin urmare, doar termenul dominant al formulei (adică an^2) deoarece ceilalţi termeni sunt relativ nesemnificativi pentru valori mari ale lui n. Ignorăm, de asemenea, şi factorul constant c, deoarece pentru numere foarte mari, factorii constanţi sunt mai puţin semnificativi decât rata de creştere în determinarea eficienţei computaţionale a unor algoritmi. Astfel ,vom spune ,de exemplu, că sortarea prin inserţie are un timp de execuţie în cazul cel mai defavorabil de $\theta(n^2)$ (pronunţat *teta de pătrat*). Vom folosi notaţia de tip θ în acest capitol cu caracter informal; va fi definită cu precizie în capitolul 2.

În mod uzual, vom considera un algoritm ca fiind mai eficient decât altul dacă timpul său de execuţie în cazul cel mai defavorabil are un ordin de creştere mai mic. Această evaluare ar putea fi incorectă pentru date de intrare de dimensiune mică, dar în cazul unor date de intrare de dimensiuni foarte mari, un algoritm de tipul $\theta(n^2)$, de exemplu, va fi executat în cazul cel mai defavorabil mult mai repede ca unul de tipul $\theta(n^3)$.