

Doina Logofătu

Algoritmi fundamentali în C++ Aplicații

- Introducere în POO
- Introducere în STL
- Greedy
- Recursivitate
- Backtracking
- Programare dinamică



68 de probleme rezolvate
252 de probleme propuse

Doina Logofătu (n. 1974) este absolventă a Facultății de Informatică, Universitatea „A.I.I. Cuza” din Iași. În 1999 a absolvit cursurile de studii aprofundate (masterat), cu lucrarea de disertație *Programarea distribuită în Java RMI, o analiză comparativă*. A predat la Liceul de Informatică „Grigore C. Moisil” din Iași, a susținut laboratoare de C/C++ la Facultatea de Informatică din Iași, a lucrat ca programator la firme de soft din Iași și München. În prezent, lucrează ca programator în München. Este coautoare a manualului de informatică pentru clasa a XI-a, profilul matematică-informatică (Petrițion, 2001), autoare a lucrărilor C++. *Probleme rezolvate și algoritmi* (Polirom, 2001), *Bazele programării în C. Aplicații* (Polirom, 2006) și a unor articole publicate în revista *GInfo*. A mai publicat în Germania volumul *Algorithmen und Problemlösungen mit C++* (Vieweg, 2006) și lucrări la conferințe internaționale.

© 2007 by Editura POLIROM

www.polirom.ro

Editura POLIROM

Iași, Bdul Carol I nr. 4, P.O. BOX 286, 700506

București, Bdul I.C. Brătianu nr. 6, et. 7, ap. 33, O.P. 37; P.O. BOX 1-728, 030174

Descrierea CIP a Bibliotecii Naționale a României:

LOGOFĂTU, DOINA

Algoritmi fundamentali în C++ / Doina Logofătu. – Iași:
Polirom, 2007

ISBN 978-973-46-0093-9

004.43 C++

Printed in ROMANIA

Doina Logofătu

**Algoritmi
fundamentali
în C++
Aplicații**

Surorii mele Monica

Cuprins

<i>Cuvânt înainte</i>	9
Capitolul 1. Algoritmi – elemente definitorii	
Geneza cuvântului algoritm	13
Definiții alternative	13
Exemple	14
De la problema spre soluție	18
Proprietățile unui algoritm	21
Algoritmica	21
Modelul de calcul RAM	22
Complexitatea algoritmilor	23
Exemplu comparativ de creștere a lui $O(g(n))$	27
Timpul real necesar unui algoritm (polinomial vs exponential)	27
Clasificarea problemelor (P , NP , NP -complete, NP -hard)	28
Probleme NP -complete	29
Problema satisfiabilității (SAT)	30
Clasa problemelor NP -hard	31
Exerciții și probleme propuse	31
Capitolul 2. Introducere în POO	
Concepțe de bază	37
Proprietăți ale programării orientate obiect	41
Crearea și distrugerea obiectelor	43
Inheritance	44
Avantajele utilizării programării orientate obiect	45
Probleme propuse	45
Capitolul 3. Introducere în STZ	
Înțeles, concepție de lucru	46
Programarea generică	49
Containere	51
Iteratori	55
Algoritmi	58
Functori	61
Încă șapte probleme rezolvate	63
Întrebări, exerciții, probleme propuse	72

Capitolul 4. Cutii ascunse	75
Descrierea problemei	77
Analiza problemei și proiectarea soluției	78
Algoritmul	79
Programul	81
Analiza programului	84
Trei mici trucuri de programare	85
Probleme propuse	86
Observații	86
Capitolul 5. Sume de puteri	89
Descrierea problemei	91
Analiza problemei. Modelarea algebrică	92
De la formula recursivă spre algoritm	93
Algoritmul	96
Programul	98
Probleme propuse	101
Capitolul 6. Greedy	103
Fundamente	105
Problema 1. Problema fracționară a rucsacului	106
Problema 2. Colorarea hărții	110
Problema 3. Calul pe tabla de sah	112
Problema 4. Arborele parțial de cost minim (algoritmul lui Kruskal)	115
Problema 5. Codificarea Huffman	119
Capitolul 7. Problema ordonării datelor	127
Descrierea problemei	129
Domeniul problemei. Definiții	129
POD și PODI sunt NP-complete	134
Algoritmi pentru POD și PODI	135
Algoritmul Random (RAN)	136
Algoritmii Exact (EX)	137
Algoritmii Greedy Min Simplified (GMS)	138
Algoritmii de mărginire inferioară (LB)	138
Detalii de implementare	140
Program (STL, parametri în linia de comandă)	143
Interpretația rezultatelor	147
Probleme propuse	153
Capitolul 8. Recursivitatea	155
Inducția matematică	157
Recursivitatea. Fundamente	163
Problema 1. Suma cifrelor și oglinditul unui număr natural	164
Problema 2. Numărul 4	166

Problema 3. Big Mod	167
Problema 4. Tortul (recursivitate liniară)	169
Problema 5. Funcția Ackermann (recursivitate imbricată – <i>compound recursion</i>)	171
Problema 6. Transformare recursivă în altă bază (din baza 10 în baza p)	173
Problema 7. Suma a două rădăcini (recursivitate ramificată)	175
Problema 8. Funcția lui Collatz (recursivitate nemonotonă)	176
Problema 9. Pătrate și pătrăjele	178
Problema 10. Pătrate (recursivitate directă)	181
Problema 11. Pătrate și cercuri (recursivitate indirectă)	190
Problema 12. Curba fulgului de zăpadă a lui Koch	194
Capitolul 9. Divide et Impera	203
Fundamente	205
Problema 1. Cel mai mare divizor comun	206
Problema 2. Turnurile din Hanoi	207
Problema 3. Integrală cu regula trapezului	209
Problema 4. <i>QuickSort</i> (sortarea rapidă)	211
Problema 5. <i>MergeSort</i> (sortarea prin interclasare)	213
Problema 6. Arbori <i>Quad</i> (<i>Quad-Trees</i>)	215
Problema 7. Transformata <i>Fourier</i> discretă (<i>DFT</i>)	220
Capitolul 10. Backtracking	225
Problema 1. Problema celor n regine	227
Observații generale cu privire la metoda <i>Backtracking</i>	232
Problema 2. Problema celor n ture pe tabla de șah	234
Problema 3. Problema turelor pe primele m linii ale unei table de șah	235
Problema 4. Problema turelor în scără pe primele m linii ale unei table de șah	236
Problema 5. Tabăra prieteniei	237
Problema 6. Partițiile unui număr natural	238
Problema 7. Referate la geografie	241
Problema 8. Toate drumurile călului pe tabla de șah	244
Problema 9. Problema fotografiei	246
Problema 10. Rostogolirea mingii	248
Problema 11. Problema taberei	250
Problema 12. Sudoku	257
Problema 13. Căsuța lui Moș Crăciun	263
Problema 14. Compactarea săbloanelor de test	265
Încă 12 probleme propuse	273
Capitolul 11. Programarea dinamică	283
Fundamente și proprietăți ale metodei	285
Problema 1. Numărarea iepurășilor	289
Problema 2. Subșir crescător maximal	293
Problema 3. Cel mai lung subșir comun (<i>LCS</i>)	296
Problema 4. Triunghi de numere	300
Problema 5. Domino	303

Problema 6. Împărțirea cadourilor	306
Problema 7. Sume asemănătoare	309
Problema 8. Scoțieni la Oktoberfest	314
Problema 9. Calul pe tabla de sah	320
Problema 10. Fluctuații la bursă	326
Problema 11. Sume de produse	329
Problema 12. Triangularizare minimală a unui poligon convex	333
Problema 13. Înmulțirea unui sir de matrice	339
Problema 14. <i>Edit-Distance</i>	344
 <i>Bibliografie</i>	353
<i>Glosar</i>	356

Cuvânt înainte

Această carte se bazează preponderent pe o parte a volumului *Algorithmen und Problemlösungen mit C++*, care a apărut în Germania (octombrie 2006). Cartea de față cuprinde noi probleme rezolvate (ex. Codificarea Huffman, Sudoku, LCS) în cadrul capitolelor corespondente, dar și noi capitole (ex. „Introducere în STL”, „Problema ordonării datelor”). O serie de programe au fost refăcute și extinse, au fost de asemenea adăugate noi figuri, imagini și tabele. Cartea cuprinde experiențe, observații și cunoștințe pe care le-am adunat în decursul a peste 10 ani. În studenție, mi-am dorit cărți care să ușureze accesul spre teorie și să conțină multe exemple practice. Ca profesoră, mi-am dorit cărți care să conțină probleme atractive, care să mărească interesul și curiozitatea elevilor pentru informatică. Ca programator, mi-am dorit cărți în care să găsesc soluții la probleme specifice. Câteodată aș lua o carte în concediu, din care să citesc mici povestiri și în același timp să repet concepte teoretice, fără să trebuiască să mă pierd în multe și complicate formalizări. Acum, când încerc să găsesc motivele pentru care e nevoie de această carte, mă gândesc la toate aceste lucruri și cred că, vrând-nevrând, am încercat să le cuprind pe toate sub același acoperiș.

Algoritmi fundamentali în C++ cuprinde în jur de 70 de probleme complet analizate și rezolvate în C++, 250 de probleme propuse și 130 de figuri și imagini. Primele trei capitoile conțin aspecte generale, teoretice despre algoritmi, programarea orientată obiect și *Standard Template Library (STL)*. Fiecare capitol începe cu o secțiune de fundamente, care face posibilă o privire de ansamblu asupra trăsăturilor metodei. Pentru fiecare problemă se descrie cum sunt construite fișierele de intrare și ieșire și este furnizat un exemplu sugestiv. În acest fel puteți verifica corectitudinea unei soluții proprii. Secțiunea *Analiza problemei și proiectarea soluției* prezintă detaliat o soluție a problemei și este urmată de *listing-ul complet* al programului C++. Programele sunt compacte și cuvintele-cheie sunt îngroșate pentru a ușura citirea lor, chiar și dacă nu sunteți familiarizați cu limbajul C++. Acestea pot fi copiate și modificate și să fie compilate cu Microsoft® Visual C++ 2005 Express Edition, oferit gratuit de Microsoft®, și încearcă să respecte standardul ANSI C++, astfel încât ar trebui să funcționeze cu orice compilator de C++. O excepție o reprezintă ultimele trei programe din capitolul 8, „Recursivitatea”, care desenează structuri fractale. Acestea folosesc biblioteca *Active Template Library (ATL)* și sunt concepute special pentru Microsoft® Windows® (pentru primele două este prezentată și o implementare Borland C++ 3.1, care funcționează și pentru DOS®). Fiecare problemă rezolvată este urmată de

exerciții și probleme propuse, care cer, de exemplu, modificarea sau extinderea programelor expuse sau scrierea altora noi, cu scop recapitulativ sau pentru a îmbunătăți aptitudinile de programare.

Toate problemele se referă la metoda și concepțele expuse la începutul capitolului. Scopul este acela de a învăța teorie prin intermediul aplicării în practică, deci la probleme concrete. Multe dintre acestea sunt clasice, cum ar fi: curba fulgului de zăpadă a lui Koch, tururile din Hanoi, parcurgerile DFS și BFS în graf, problema celor n regine, colorarea hărții, cel mai lung subșir comun (LCS), înmulțirea unui sir de matrice și distanța de editare. Olimpiadele județene și naționale, precum și concursurile de programare *Association for Computing Machinery (ACM)*, *International Olympiad in Informatics (IOI)*, *Central-European Olympiad in Informatics (CEOI)* au fost surse de inspirație pentru numeroase probleme din carte.

Ca răsplătă pentru sărăuința dumneavoastră, veți găsi la sfârșitul fiecărui capitol câte o fotografie surpriză, cum ar fi: fan bavarez la Campionatul Mondial de Fotbal (Germania, 2006), arbori în Ottawa, case oglindite în Lübeck sau statuie în Bremen.

Dacă doriți, vă rog să îmi scrieți observațiile și propunerile dumneavoastră la adresa doinabooks@yahoo.com, lucru pentru care vă mulțumesc anticipat.

Lectură și învățare plăcute!

Doina Logofătu
München, februarie 2007

1 **Capitolul** **Algoritmi – elemente definitorii**

- Geneza cuvântului *algoritm*
- Definiții alternative
- Patru exemple de algoritmi (algoritmul lui Euclid, „ciurul lui Eratostene”, căutarea binară, prepararea prăjiturii Tiramisu)
- De la problemă spre soluție
- Proprietățile unui algoritm
- Algoritmica
- Modelul de calcul *RAM*
- Complexitatea algoritmilor
- Optimalitatea și reducerea algoritmilor. Exemple
- Exemplu comparativ de creștere a lui $O(g(n))$
- Timpul real necesar unui algoritm (polinomial vs exponentiațial)
- Clasificarea problemelor (*P*, *NP*, *NP*-complete, *NP-hard*)
- Probleme *NP*-complete
- Problema satisfiabilității (*SAT*)
- Clasa problemelor *NP-hard*
- Exerciții și probleme propuse

Inceputul este jumătatea întregului.
Aristotel

Deglă computerele și aplicațiile variate corespunzătoare lor sunt un fenomen exploziv ce a devenit predominant în ultimii 20 de ani, o mare parte dintre concepții ce stau la baza acestor instrumente soft datează de foarte mult timp, chiar din Antichitate, și ele s-au dezvoltat de-a lungul timpului.

Geneza cuvântului algoritm

Cuvântul *algoritm* provine de la numele matematicianului, geografului și astrologului arab *Abu Ja'far Muhammad ibn Musa Al-Khwarizmi* (780?-850?), care a trăit în Bagdad, capitala Irakului. Cea mai importantă lucrare a sa este *Hisab al-jabr w'al-muqabala*, iar titlul va conduce și la geneza cuvântului *algebra*, utilizat pentru a defini această ramură a matematicii. El a lucrat la Casa Înțelucirii din Bagdad, sub patronajul direct al califului Al-Ma'mun, unde traducea manuscrisele științifice grecești și studia algebra, geometria și astrologia. Vechii hinduși, greci, babilonieni, romani și chinezi foloseau termenul *algorism* pentru operații aritmetice simple.



Definiții alternative

- Un algoritm este o mulțime de reguli ce se pot aplica în cadrul procesului de construcție a soluției și pot fi executate fie „de mână”, fie de o mașină.
- Un algoritm este o secvență de pași care transformă mulțimea datelor de intrare în datele de ieșire, rezultându-ne date.
- Un algoritm este o secvență de operații executate cu date ce trebuie organizate în structuri de date.
- Un algoritm este abstractizarea unui program care trebuie executat pe o mașină fizică (model de calcul).
- Un algoritm pentru o problemă dată este o mulțime de instrucțiuni care garantează găsirea unei soluții corecte pentru orice instanță a problemei, într-un număr finit de pași.

Exemple

Algoritmul lui Euclid

Nu există dovezi sigure pentru nimic din ceea ce se afirmă astăzi despre Euclid, iar uneori se pune la indoială chiar și existența acestuia. Se presupune că a trăit în jurul anului 300 i.Ch. și că a fost un matematician grec, că a predat la Alexandria, în Egipt. Se pare că el a scris *Elementele*, o lucrare didactică exemplară, care sintețizează cunoștințele de la acea vreme ale matematicilor grecești. În a șaptea carte se găsește algoritmul lui Euclid:



Euclid

Presupunem că a și b sunt două numere naturale, $a \geq b$, și că dorim aflarea celui mai mare divizor comun al lor. Considerăm $a_1 = a$ și $b_1 = b$ și definim perechile (m, n) , astfel încât $a_i = m_i b_i + n_i$, $0 \leq n_i < b_i$. Mai departe, vom considera $a_{i+1} = b_i$ și $b_{i+1} = n_i$. Atunci există un număr natural k , astfel încât $n_k = 0$. Mai mult, dacă $n_k = 0$, $\text{cmmdc}(a, b) = n_{k-1}$.

ALGORITM_EUCLID

1. Citește $a, b \in \mathbb{N}$, $a \geq b > 0$
2. $a_1 \leftarrow a$, $b_1 \leftarrow b$, $i \leftarrow 1$
3. While ($b_i \neq 0$) Do
 - 3.1. $a_{i+1} \leftarrow b_i$
 - 3.2. $b_{i+1} \leftarrow n_i$ ($= a_i \bmod b_i$)
 - 3.3. $i \leftarrow i + 1$
4. $\text{ggT}(a, b) = n_{i-1}$

END_ALGORITM_EUCLID

Exemple:

$a = 294$, $b = 11$	$a = 2521$, $b = 338$
$294 = 3 \times 97 + 63$	$2521 = 7 \times 338 + 155$
$97 = 1 \times 63 + 14$	$338 = 2 \times 155 + 28$
$63 = 4 \times 14 + 7$	$155 = 5 \times 28 + 15$
$14 = 2 \times 7 + 0$	$28 = 1 \times 15 + 13$
	$15 = 1 \times 13 + 2$
	$13 = 6 \times 2 + 1$
	$2 = 2 \times 1 + 0$

Algoritmul 2. Clurul lui Eratostene

Eratostene (?276-194 î.Chr., Cyrene acum Libia, Africa de Nord) a fost primul mare geograf al lumii antice și este considerat fondatorul geografiei fizice și matematice. A lucrat mult timp la celebra bibliotecă din Alexandria din Egipt și a fost, de asemenea, interesat de geometrie și de numere prime. El a măsurat primul circumferință Pământului (care s-a dovedit ulterior corectă) și a creat o hartă a lumii, pe care a desenat linii paralele. În plus, a mai scris un poem în versuri, *Hermes*, despre fundamentele astronomiei; a sugerat că anii bisecți se repetă la fiecare 4 ani; a introdus metoda afilării numerelor prime numită „sita lui Eratostene”. În memoria sa, există azi un crater pe Lună care îl poartă numele.



Eratostene

Definiție. Un număr prim este un număr natural mai mare decât 1 și care se divide numai cu 1 și cu el însuși. Un număr natural care se divide cu un alt număr decât 1 și el însuși se numește număr compus. „Sita (ciurul) lui Eratostene” este un algoritm care identifică toate numerele prime mai mici sau egale cu un număr natural n dat.

Exemplu:

ALGORITM_SITA_ERATOSTENE

1. Se scriu în ordine toate numerele de la 1 la n . (Le vom elimina succesiv pe cele compuse prin marcarea lor. Înțîl, toate numerele sunt nemarcate.)
 2. Marchează 1 ca număr special (nu este nici prim, nici compus)
 3. $k \leftarrow 1$
 4. Do
 - 4.1. $m \leftarrow$ primul număr mai mare decât k , dintre cele care nu au fost marcate până la k
 - 4.2. marchează numerele $2m, 3m, 4m, \dots$ din listă ca fiind compuse (mai întâi se vor marca toți multiplii lui 2, apoi multiplii lui 3 rămasi, ...)
 - 4.3. este un număr prim. adaugă-l în lista de numere prime
 - 4.4. $k \leftarrow m$
 5. While ($k \geq \sqrt{n}$)
 - 5.1. Adaugă numerele rămase nemarcate în lista numerelor prime
 - 5.2. Afisează lista numerelor prime până la n
- END_ALGORITM_SITA_ERATOSTENE

eliminarea multiplilor lui 2	eliminarea multiplilor lui 3
(2) 3 5 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48	(2) (3) 5 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48
eliminarea multiplilor lui 5	numerele prime mai mici decât 49
(2) (3) 5 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48	(2) (3) (5) 7 8 9 10 11 (13) 14 15 16 (17) 18 (19) 20 21 22 (23) 24 25 26 27 28 (29) 30 31 (32) 33 34 35 36 (37) 38 39 40 41 (42) (43) 44 45 46 (47) 48

Cititi și implementarea din capitolul 3, problema 10.

Algoritm 3. Căutarea binară

Presupunând că se dă ca date de intrare un sir finit ordonat crescător de elemente a_1, a_2, \dots, a_n și un element x , să se determine prima poziție a lui x în sir, dacă acesta există, altfel, să se returneze valoarea 0.

Informația că sirul de elemente date este deja ordonat ne conduce la determinarea unui algoritm, prin micșorarea succesivă a intervalului de căutat (de exemplu, dacă ultimul element a_n este mai mic decât x , ne oprim deja după o primă comparație și afișăm că x nu se află în sir). Vom vedea mai târziu de ce acest algoritm este în practică mai eficient decât unul în care s-ar efectua căutarea liniară, adică cercetarea tuturor elementelor.

ALGORITM_CĂUTARE_BINARĂ

1. *Initializare*
2. $i \leftarrow 1, j \leftarrow n$
3. **While** ($i < j$) **Do**
 - 3.1. $m \leftarrow (i + j)/2$
 - 3.2. If ($x > a_m$) Then $i \leftarrow m + 1$
 - 3.3. Else $j \leftarrow m$**End_While**
4. If ($x = a_i$) Then $poz \leftarrow i$
5. Else $poz \leftarrow 0$
6. *Scrie poz*

END_ALGORITM_CĂUTARE_BINARĂ

Ilustrăm în continuare aplicarea algoritmului pentru sirul ordonat de 17 litere $a \in d f g h i l m o p r s u v x z$, în care ne propunem să căutăm litera j .

$a_1, \dots, a_n = a c d f g h j l m o p r s u v x z$	
$x = j$	
$i \leftarrow 1, j \leftarrow 17$	
$a c d f g h j l m o p r s u v x z$	$a c d f g h j l m o p r s u v x z$
$i \quad \quad \quad j$	$i \quad \quad \quad j$
$i = 1, j = 17, m \leftarrow 9$ (3.2. sau 3.3. ?)	$i = 1, j = 9$ (pas 3.3.), $m \leftarrow 5$ (3.2. sau 3.3. ?)
$a c d f g h j l m o p r s u v x z$	$a c d f g h j l m o p r s u v x z$
$i \quad \quad \quad j$	$i \quad \quad \quad j$
$i = 5, j = 9$ (pas 3.2.), $m \leftarrow 7$ (3.2. sau 3.3. ?)	$i = 7, j = 9$ (pas 3.2.), $m \leftarrow 8$ (3.2. sau 3.3. ?)
$a c d f g h j l m o p r s u v x z$	$a c d f g h j l m o p r s u v x z$
$i \quad \quad \quad j$	$(i = j)$
$i = 7, j = 8$ (pas 3.3.), $m \leftarrow 7$ (3.2. sau 3.3. ?)	$i = 7, j = 7$ Stop

Pasul 3.2 din algoritm ne indică faptul că dacă numărul x căutat se situează la dreapta elementului din mijlocul intervalului cercetat (m este mijlocul intervalului $[i, j]$), atunci el va trebui căutat în intervalul $[m + 1, j]$, altfel se va efectua căutarea în intervalul $[i, m]$, capetele i , respectiv j modificându-se corespunzător.

Algoritm 4. Prepararea prăjiturii Tiramisu



Ingredienti: un pachet de biscuiți (de campanie sau Löffelbiskuits), 500 g Mascarpone (specialitate de brânză italiană dulce și moale), 4 ouă, 150 g zahăr, un pahar mic (10 ml) de lichior Amaretto, un pachețel de pudră de gelatină, 3 ml cacao, o cană de cafea îndulcită.

La prima vedere, pare că nu există nici o legătură între bucătărie și algoritmică, dar, dacă încercăm să facem o paralelă, vom observa că lucrurile nu stau deloc așa. Rețeta de preparare este, de fapt, un algoritm care, pe baza unor date de intrare (ingredientele), produce datele de ieșire cerute (prăjitura). Pasii se execută succesiv și ordinea de execuție a unor pași specifici este foarte importantă, exact cum și o serie de instrucțiuni (pași) se pot interschimba. De exemplu, nu putem executa pasul 6 înaintea pasului 5, dar putem, în schimb, executa pasul 7 înaintea pasului 5. Observăm cum ingredientele sunt combinate și transformate pentru a ajunge la îndeplinirea obiectivului final – obținerea desertului. Se poate descrie și un alt algoritm care va conduce la același rezultat (de exemplu, dacă biscuiții sunt îmbăbiți succesiv cu cafea cu ajutorul unei lingurite, după ce au fost așezăți în tavă). Ca un exercițiu de

imaginărie, putem chiar să ne gândim și la construirea unei mașini care să prepare automat Tiramisu.

ALGORITM_PREPARARE_TIRAMISU

1. Start preparare.
2. Răsturnați cafeaua într-o farfurie de supă.
3. Înmulțiați biscuiții, ținându-i puțin pe fiecare în farfurie cu cafea (ei trebuie să fie pe jumătate umede).
4. Așezați biscuiții unul lângă altul în tavă, astfel încât aceștia să „tapeteze” tava.
5. Separați ouăle.
6. Amestecați gălbenușurile, zahărul și lichiorul Amaretto cu mixerul, până se obține o cremă uniformă.
7. Adăugați în crema astfel obținută specialitatea Mascarpone și gelatină și amestecați în continuare.
8. Bateți cu mixerul separat și albușurile până când spuma obținută devine tare (prin întoarcerea bolului, aceasta nu cade).
9. Răsturnați albușurile bătute în crema cu gălbenușuri și amestecați totul cu mixerul.
10. Acoperiți uniform stratul de biscuiți cu o jumătate din cremă. Peste stratul de cremă așezați încă un strat de biscuiți (înmulțați în cafea pe lungime) și peste acest st. o cîteva bule uniform restul de cremă.
11. Așezați tava în frigider pentru cel puțin trei ore (cel mai bine, peste noapte).
12. Înainte de a se servi, se presară cacao pudră.
13. Sfărșit preparare. Poftă bună!

END ALGORITM_PREPARARE_TIRAMISU

De la problemă spre soluție

Problemele din sfîrșitul informaticii actuale, indiferent dacă sunt de dimensiune redusă sau adevarate produse gigant, au la bază o serie de idei și caracteristici comune. Comportarea internă în cadrul ciclului de viață al unui program se bazează întotdeauna pe două elemente primordiale: reprezentarea și transformarea.

- *Reprezentarea* se referă la o codificare concretă a informației (biți, numere, cuvinte, înregistrări).
- *Transformarea* se referă la modificarea succesivă a stării, prin derularea unor pași (rețetă, program, algoritm), cu scopul de a găsi rezultatele așteptate.

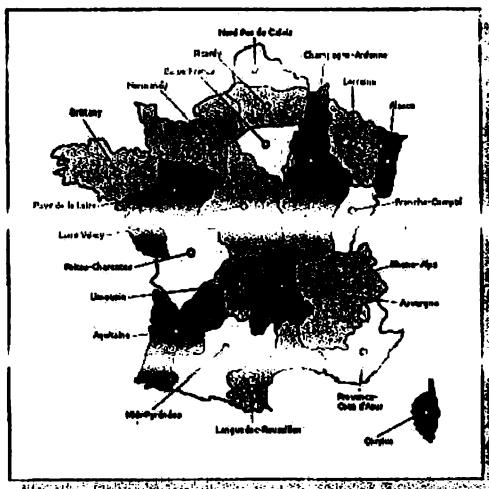
Codificarea simbolică a informației (*reprzentarea*) furnizează o modalitate de comunicare între lumea reală a problemei și lumea „imaginată” a computerului. Se va crea astfel o „mapare”, funcție cu ajutorul căreia entitățile reprezentative din cadrul problemei de rezolvat sunt codificate prin valori înțelese de calculator. Această etapă de codificare este foarte importantă, deoarece de ea depind corectitudinea soluției, complexitatea structurilor alese și a algoritmului.

Rezolvarea unei probleme de informatică presupune construirea a trei procese, ce se întrepătrund și se execută secvențial:

- găsirea unei reprezentări simbolice a informației, astfel încât să poată fi manipulate ulterior;
- formularea algoritmului care, utilizând aceste reprezentări, va conduce la rezultatele dorite;
- construirea cadrului specific ce permite ca acești algoritmi să fie implementați într-o manieră corespunzătoare (de exemplu, scrierea codului într-un limbaj de programare).

Exemplul 1. Problema colorării hărții (*Map Colouring Problem*). Dată fiind o hartă cu țări, ne propunem să găsim o modalitate de a colora fiecare țară, astfel încât orice două țări vecine să fie colorate diferit (culorile se vor alege dintr-o paletă de culori dată).

De exemplu, harta Franței va trebui desenată astfel încât oricare două regiuni vecine să fie colorate diferit, folosind culorile roșu, galben, verde și albastru. Problema apare în forma ei practică și, pentru a o rezolva, trebuie să o codificăm în aşa fel încât să permitem modelarea ei cu ajutorul unui algoritm. Intuitiv, ne dăm seama că primul pas va fi atașarea unui număr fiecărei regiuni, de la 1 la numărul acestora n . Mai departe, va trebui să manipulăm și informațiile de vecinătate, adică am putea construi o matrice pătratică A de dimensiune $n \times n$, în care celula $A(i, j)$ va primi valoarea 1 sau 0, după cum regiunile i și j sunt – respectiv, nu sunt – vecine. Vom codifica și culorile cu numerele de la 1 la m , presupunând că dispunem de m culori. În felul acesta, am realizat o



modelul folosit este *RAM* (*Random Access Machine*), dar există și alte modele de calcul, precum: *PRAM*, *Mașini Turing*, automate finite, circuite booleene, automate celulare etc. Odată modelul definit, algoritmul poate fi descris folosind un limbaj simplu, asemănător ca sintaxă; de exemplu, *Basic*, *Pascal*, *C*, *C++*, *Java* etc.

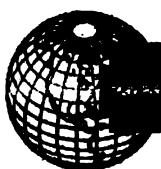
Modelul de calcul RAM

Proiectarea unui algoritm independent de mașină depinde de un computer ipotetic numit *Random Access Machine* (*RAM*). Potrivit acestui model, computerul ipotetic satisface:

- Fiecare operație „simplă” (+, -, *, /, %, =, *if/daici*, *call/apelaazi*) consumă o unitate de timp.
- Ciclurile și metodele (funcții, proceduri) nu sunt considerate operații simple. Acestea sunt compuse din mai multe operații de un singur pas. Nu ar avea sens să atribuim metodei *sortare*, de exemplu, o unitate de timp, când sortarea unui milion de elemente va consuma mult mai mult timp decât sortarea a 10 elemente. Timpul necesar pentru a executa un ciclu sau o metodă (subprogram) depinde de numărul de iterări, respectiv de natura metodei.
- Fiecare acces la memorie se execută într-o unitate de timp și avem la dispoziție atâta memorie câtă este necesară. Modelul *RAM* nu face diferență dacă anumite date sunt pe disc sau în *cache* (memoria volatilă), ceea ce simplifică analiza algoritmului.

Conform cu modelul *RAM*, se vor cauza numărul de pași (unități de timp) necesari pentru instanța unei probleme. Presupunând că *RAM* execută un anumit număr de pași pe secundă, se poate ușor determina timpul real necesar. Una dintre nemulțumiri ar putea fi faptul că modelul este mult prea simplu, că unele presupuneri sunt prea generale pentru a fi aplicate în practică. De exemplu, pentru înmulțirea a două numere, este nevoie, de obicei, de mai mult timp decât pentru adunarea lor, ceea ce contrazice ipoteza de la punctul anterior. În realitate, există și modeluri mai complexe, în funcție de locul de stocare a datelor, în *cache* (memoria volatilă) sau pe disc, ceea ce contrazice cea de-a treia presupunere. Dar chiar și cu aceste neajunsuri, modelul *RAM* este un instrument excelent pentru a înțelege cum se comportă un algoritm în realitate, el reflectând cum urmărește computerul, dar rămânând destul de simplu pentru a putea fi manipulat și înțeles.

Fiecare model are un interval-limită în care este util. Să considerăm ca exemplu *modelul Pământului plat*. Se poate afirma că acesta este un model incorrect, pentru că este dovedit faptul că Pământul este rotund. Dar, pentru a construi fundația unci case, modelul Pământului plat este util și poate fi utilizat ca atare. Este mult mai ușor de lucrat practic cu un astfel de model decât cu un model sferic. Tot astfel se întâmplă și cu modelul de calcul *RAM*. El este,



de fapt, o abstractizare care, în general, devine foarte utilă pentru a explica performanța unui algoritm, într-un mod independent de mașină.

Complexitatea algoritmilor

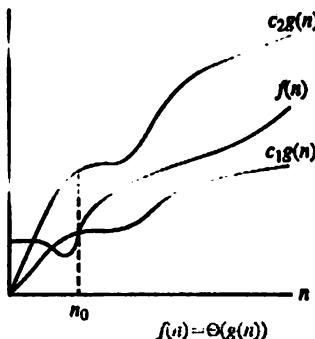
Prin **complexitatea** unui algoritm înțelegem de fapt **costul**, măsurat cu ajutorul unor anumiti parametri (timpul de execuție, memoria necesară, numărul anumitor operații etc.). Pentru a calcula complexitatea unui algoritm, avem nevoie să decidem care sunt acești parametri și să găsim o funcție cost corespunzătoare.

Dacă fiind că resursele de memorie sunt, în practică, foarte mari, deducem că timpul de execuție al unui algoritm este parametrul de bază. Complexitatea timp poate fi calculată în funcție de numărul de operații elementare (unități de timp) necesare atunci când datele de intrare au o anumită dimensiune n . Așa cum am văzut mai sus, la descrierea modelului RAM, aceste operații de bază pot fi comparații (în număr foarte mare, de exemplu, în cazul unui algoritm de căutare), asignări (de exemplu, în cazul unui algoritm de sortare), adunări, înmulțiri, împărțiri, modulo, apeluri de metodă etc.



Notățile Θ , O și Ω . În practică, sunt folosite o serie de notații care sunt utile pentru analiza performanței și a complexității unui algoritm. Aceste notații mărginesc valoarea unei funcții f date cu ajutorul unor constante și al altrei funcții. Prezentăm în continuare cele trei notații cunoscute în acest sens.

- **Notația Θ (categorie constantă – same order).** Spunem că $f(n) = \Theta(g(n))$, dacă există constantele pozitive c_1 , c_2 și n_0 , astfel încât pentru toate numerele naturale n de la dreapta lui n_0 valoarea lui $f(n)$ se află întotdeauna între $c_1 \times g(n)$ și $c_2 \times g(n)$ inclusiv.



Notația Θ : $\exists c_0, c_1, c_2$, a.î. $c_1 \times g(n) \leq f(n) \leq c_2 \times g(n)$, $\forall n \geq n_0$.

codificare simbolică („mapare”) a informației reale, utilizând instrumente matematice, prelucrabile cu ajutorul computerului. Soluția ar trebui să fie un vector de „culori” (numere) $S = \{c_1, c_2, \dots, c_n\}$, $c_i \in \{1, \dots, m\}$, cu proprietatea că, dacă $A(i, j) = 1$, atunci $c_i \neq c_j$. Odată stabilită această reprezentare simbolică a informației reale, va trebui să trecem la elaborarea unui algoritm care, pe baza matricei A și a numărului de culori m , să ne furnizeze o soluție S . De fapt, reprezentarea problemei este un graf neorientat cu matricea de adiacență A , în care va trebui să colorăm vârfurile, astfel încât oricare două vârfuri vecine să fie colorate diferit.

<p>8 regiuni, 3 culori Se cere determinarea unei colorări a regiunilor utilizând cele 3 culori, astfel încât oricare două regiuni vecine să fie colorate diferit.</p> $A = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$	<ol style="list-style-type: none"> 1. Construirea grafului atașat hărții și a matricei de adiacență în care $A(i, j) = 1$ dacă și numai dacă $i \neq j$ și regiunile i, j sunt vecine. 2. Observăm că matricea de adiacență este simetrică: $A(i, j) = A(j, i)$, $\forall i, j \in \{1, 2, \dots, n\}$. 3. Culorile posibile vor fi c_1, \dots, c_m. 4. Soluția va trebui să fie de forma $S = \{c_1, c_2, \dots, c_n\}$, $c_i \in \{1, \dots, m\}$, cu semnificația că c_i este culoarea aleasă a vârfului i al grafului.
--	--

Exemplul 2. Variantă a problemei orarului (*Schedule Problem*). Concret, presupunem că dorim să distribuim săliile, profesorii, grupele de studenți pentru a realiza orarul unei facultăți. Codificarea informației practice este mult mai dificilă în acest caz decât cea din exemplul anterior, această „mapare” nu se mai produce atât de natural și accesibil. Va trebui să codificăm cu numere de la 1 la n_1 profesorii, numere de la 1 la n_2 toate

grupele de studenți (plus fiecare student și apartenența sa la grupe), la fel locațiile (săli, amfiteatre, laboratoare etc.), cursurile posibile. Vor trebui stabilite diverse funcții (caracteristici) care trebuie să fie satisfăcute de o eventuală soluție (fiecare curs are o durată și se poate desfășura doar în anumite săli, o grupă are doar anumite cursuri depinzând de an și specializare, un profesor predă doar anumite cursuri cu anumite grupe etc.) și care trebuie, de asemenea, accesibil codificate. Vor trebui definite și o serie de restricții, astfel încât o posibilă soluție să poată fi verificată matematic (un profesor sau un student nu poate fi în două locuri simultan, într-o anumită sală nu se pot suprapune activități diferite etc.). Numai când are loc definirea în termeni matematici a acestor entități și caracteristici, iar forma soluției este complet realizată, se poate trece la elaborarea unui algoritm pentru a găsi posibilele soluții.

Proprietățile unui algoritm

- Trebuie să posedă date de intrare (*input data*).
- Trebuie să furnizeze date de ieșire (*output data*).
- *Determinismul* (la executarea oricărui pas, trebuie să cunoaștem succesorul acestuia).
- *Corectitudinea* datelor de ieșire (a rezultatelor) în raport cu datele de intrare.
- *Finitudinea* (pentru orice set de date de intrare posibile ale problemei, soluția este furnizată într-un număr finit de pași).
- *Eficiența* (furnizarea soluției trebuie să fie realizată prin consumul eficient al resurselor timp, spațiu etc.).
- *Generalitatea* (algoritmul este aplicabil unei clase de probleme, cele ale căror date de intrare satisfac anumite condiții).

Algoritmica

Algoritmica este ramura informaticii care se ocupă cu proiectarea și analizarea algoritmilor și sărutării specifice implementării cu ajutorul computerului.

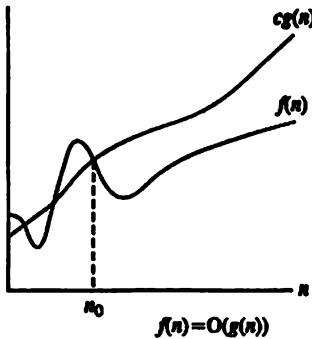
1. Prin proiectare (design) înțelegem două etape:

- descrierea algoritmului utilizând un pseudolimbaj (schemă logică, pseudocod, descriere cuprinzătoare sugestivă pe o foaie de hârtie etc.);
- realizarea corectă a algoritmului, indiferent de datele de intrare posibile introduse, rezultatul satisfac cerința problemei.

2. Analiza algoritmului se referă la evaluarea performanței acestuia, adică a timpului necesar pentru a determina soluția, dar și a altor aspecte, cum ar fi spațiul de memorie folosit, optimizările etc.

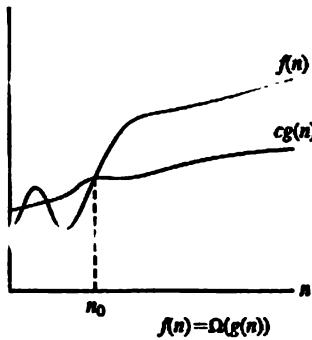
Începutul trebuie să fie alegerea sau definirea unui model de calcul, adică a unui formalism matematic ce descrie interacțiunea dintre componentele sistemului și furnizează abstractizări utile pentru concepte precum *temp* sau *concurență*. De obicei,

- Notăția O (mărginire superioară – *upper bound*).** Spunem că $f(n) = O(g(n))$, dacă există o constantă pozitivă n_0 , astfel încât, pentru toate numerele n de la dreapta lui n_0 , valoarea lui $f(n)$ se află întotdeauna sub $c \times g(n)$.



Notăția O: $\exists n_0, c, \text{ a.î. } f(n) \leq c \times g(n), \forall n \geq n_0$.

- Notăția Ω (mărginire inferioară – *lower bound*).** Spunem că $f(n) = \Omega(g(n))$, dacă există constantele pozitive n_0 și c , astfel încât, pentru toate numerele n mai mari decât n_0 , valoarea lui $f(n)$ este mai mare decât $c \times g(n)$.



$\forall n_0 \in \mathbb{N}, \exists c > 0, \Omega(g(n)) \leq f(n) \leq cg(n), \forall n \geq n_0$

Optimalitatea și reducerea algoritmilor. Exemple

Complexitatea unui algoritm este deci o funcție $g(n)$ care limitează superior numărul de operații (la execuție) necesare pentru dimensiunea n a problemei. Există două interpretări ale limitei superioare:

- a) complexitatea în cazul cel mai defavorabil (*worst-case complexity*): timpul de execuție pentru orice dimensiune dată va fi mai mic decât limita superioară, exceptând câteva dimensiuni ale problemei, unde este atins un maxim;
- b) timpul de execuție pentru orice dimensiune dată va fi media numărului de operații pentru toate instanțele posibile ale problemei.

Deoarece este dificil să se estimeze o comportare statistică ce depinde de dimensiunea intrării, de cele mai multe ori este folosită prima interpretare, cea a cazului celui mai defavorabil. În majoritatea cazurilor, complexitatea lui $f(n)$ este aproxiată de familia sa $O(g(n))$, unde $g(n)$ este una dintre funcțiile: n (complexitate liniară), $\log(n)$ (complexitate logaritmică), n^a , $a \geq 2$ (complexitate polinomială), n^α (complexitate exponențială), $n!$ (complexitate factorială). O altă definiție sugestivă echivalentă este cea folosind limitele din analiza matematică: dacă limita funcției $f(n)/g(n)$ există și este finită, atunci $f(n)$ are ordinul $O(g(n))$.

Exemple:

1. $\lim (\text{sqrt}(n)/n) = 0 \rightarrow \text{sqrt}(n) = O(n)$
2. $\lim (n/\text{sqrt}(n)) = \text{infinิต} \rightarrow n$ nu este $O(\text{sqrt}(n))$
3. $\lim (n/2n) = \frac{1}{2} \rightarrow n = O(2n)$
4. $\lim (2n/n) = 2 \rightarrow 2n = O(n)$

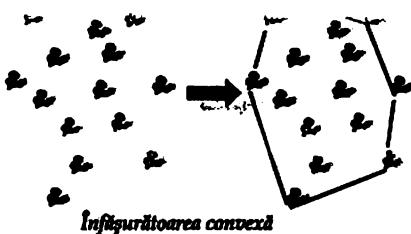
Pentru n suficient de mare au loc inegalitățile: $\log(n) < n < n \times \log(n) < n^2 < n^3 < 2^n$, ceea ce implică $O(\log(n)) < O(n) < O(n \times \log(n)) < O(n^2) < O(n^3) < O(2^n)$.

Exemple: $10n + 5 \log(n) + 8 \in O(n)$, $8n \in O(n^2)$, $65 \in O(1)$, $n^{1000} \in O(2^n)$.



Intersecția a n segmente

Odată determinată complexitatea unui algoritm, apare problema stabilirii dacă acest algoritm este *optimal*. Un algoritm pentru o problemă dată este optimal atunci când complexitatea sa atinge o limită inferioară a tuturor algoritmilor ce rezolvă aceeași problemă. De exemplu, algoritmul care rezolvă problema „intersecției a n segmente” va execuția cel puțin n^2 operații în cazul cel mai nefavorabil și există un algoritm cu această complexitate, deci vom afirma că problema are o complexitate $\Omega(n^2)$.



Infișarea convexă

Reducerea este o altă tehnică de a estima complexitatea unei probleme prin transformarea acesteia într-o problemă echivalentă. De exemplu, presupunând că știm limita inferioară a unei probleme A și că ne dorim estimarea limitei inferioare a

unei probleme *B*. Dacă putem transforma *A* în *B* prin intermediul unui pas de transformare al cărui cost este mai mic decât rezolvarea lui *A*, atunci *B* are aceeași limită ca și *A*. Ca un exemplu, putem considera problema determinării înfășurătorii convexe, care se reduce la sortarea punctelor date (complexitate: $\Theta(n \times \log(n))$).

Exemplu: $O(n) + O(\log(n)) = O(n)$ ($\log(n) < n$), $O(n!) + O(n^3) = O(n!)$ ($n^3 < n!$)).

Exemplul 1. Analiza complexității. Considerăm următorul algoritm:

1. Citește *n*
2. *dif* $\leftarrow 0, *sum* $\leftarrow 0, *k* $\leftarrow 0$$$
3. While (*k* $< n) Do

 - *sum* \leftarrow *sum* + 1
 - *dif* \leftarrow *dif* - 1
 - *k* \leftarrow *k* + 1
 End_While$
4. *k* $\leftarrow 0$
5. While (*k* $< 3n) Do

 - *sum* \leftarrow *sum* - 1
 - *k* \leftarrow *k* + 1
 End_While$
6. Scrie *sum*, *dif*

Analiza complexității:

Pasul 2 are nevoie de 3 unități de bază.

Ciclul de la pasul 3 va executa $2n$ adunări, n scăderi și $3n$ atribuiriri, deci în total $6n$ unități de timp.

Pasul 4 are nevoie de o unitate.

Pasul 5 face $2 \times 3n$ atribuiriri, $3n$ scăderi și $3n$ adunări, deci în total $12n$ unități de bază.

$$\text{Total: } 3 + 6n + 1 + 12n = 18n + 4$$

Aceasta înseamnă $O(n)$.

Exemplul 2. Analiza complexității și a eficienței. Vom face o analiză comparativă a trei algoritmi ce rezolvă aceeași problemă: calculul sumei $1 + 2 + \dots + n$. În acest scop, putem scrie următorii algoritmi:

Analiza eficienței algoritmilor

Algoritm A	Algoritm B	Algoritm C
$sum \leftarrow 0$ pentru (<i>i</i> $\leftarrow 1, n) sum \leftarrow sum + i $	$sum \leftarrow 0$ pentru (<i>i</i> $\leftarrow 1, n) { pentru (j \leftarrow 1, i) sum \leftarrow sum + j } $	$sum \leftarrow n * (n + 1) / 2$
$n + 1$ asignări, n atribuiriri	$1 + n(n + 1)/2$ asignări, $n(n + 1)/2$ atribuiriri	1 adunare, 1 înmulțire, 1 diviziune, 1 atribuire
Total: $2n + 1$	Total: $n^2 + n + 1$	Total: 4
$O(n)$	$O(n^2)$	$O(1)$

Deducem că algoritmul cel mai performant este C (cunoștințele de matematică sunt importante în conceperea algoritmilor!), fiind și algoritmul optimal pentru aflarea acestei sume, iar cel mai ineficient este B.

Exemplu comparativ de creștere a lui $O(g(n))$

Rata comparativă de creștere a funcției $O(g(n))$

n	$\log(\log n)$	$\log n$	$(\log n)^2$	n	$n \log n$	n^2	2^n	$n!$
10	2	3	11	10	33	10^2	10^3	10^5
10^2	3	7	44	100	664	10^4	10^{30}	10^{44}
10^3	3	10	99	1000	9966	10^6	10^{301}	10^{1435}
10^4	4	13	177	10.000	132.877	10^8	$10^{3.010}$	10^{19335}
10^5	4	17	276	100.000	1.660.964	10^{10}	$10^{30.103}$	$10^{243.338}$
10^6	4	20	397	1.000.000	19.931.569	10^{12}	$10^{301.000}$	$10^{2.933.369}$

Alte complexități:

- Se poate demonstra că **algoritmul lui Euclid** are complexitatea $O((\log a)(\log b))$.
- Complexitatea pentru **căutarea liniară** este $O(n)$, deoarece, în cel mai defavorabil caz, trebuie să parcurgem toată secvența de numere, deci complexitatea este liniară.
- Pentru **căutarea binară**, complexitatea este $O(\log n)$; în cazul cel mai nefavorabil avem nevoie de $(1 + \log n)$ operații pentru a găsi elementul x .
- Pentru **problema colorării hărților**, unde avem un graf cu n vârfuri și trebuie atribuite m culori, în cazul cel mai defavorabil, vor trebui generate toate combinațiile (căutare exhaustivă), care sunt m^n (numărul funcțiilor definite pe o mulțime cu n elemente cu valori într-o mulțime cu m elemente), deci complexitatea este exponentială: $O(m^n)$.

Timpul real necesar unui algoritm (polinomial vs exponențial)

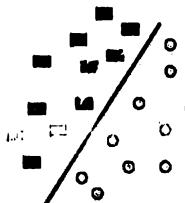
Următoarea tabelă arată cum se modifică timpul real necesar sistemului de calcul atunci când trăbunul crește de la 1 la 1000 pagini și la 1 milion de documente obligează la 1 bilion de pași/secundă. O microsecundă este a milioana parte dintr-o secundă și o milisecundă este o miile dintr-o secundă.

**Analiza comparativă a timpului real necesar
(algoritmi polinomiali vs algoritm exponențial)**

n	$T(n) = n$	$T(n) = n \lg(n)$	$T(n) = n^2$	$T(n) = n^3$	$T(n) = 2^n$
5	0.005 microsec.	0.01 microsec.	0.03 microsec.	0.13 microsec.	0.03 microsec.
10	0.1 microsec.	0.03 microsec.	0.1 microsec.	1 microsec.	1 microsec.
20	0.02 microsec.	0.09 microsec.	0.4 microsec.	8 microsec.	1 milisec.
50	0.05 microsec.	0.28 microsec.	2.5 microsec.	125 microsec.	13 zile
100	0.1 microsec.	0.66 microsec.	10 microsec.	1 milisec.	4×10^{13} ani

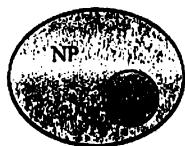
Remarcăm că, pentru $n \geq 50$, timpul de calcul pentru $T(n) = 2^n$ devine prea mare pentru a fi și practic. Chiar dacă se mărește de un milion de ori viteza sistemului, pentru $n = 100$ tot va fi nevoie de 40.000.000 de ani pentru a afla răspunsul.

Clasificarea problemelor (*P*, *NP*, *NP-complete*, *NP-hard*)



Pe lângă determinarea complexității timp, ne putem referi și la clasificarea problemelor după criteriul dat de faptul că ele sunt „greu” sau „ușor” de rezolvat (accesibile sau inaccesibile). Această schemă de clasificare include clasele *P* și *NP*, iar „*NP-complete*” și „*NP-hard*” sunt subclase ale clasei *NP*. În vederea stabilirii unor caracteristici formale relativ la mulțimea problemelor posibile, a fost mai întâi definită clasa *problemelor de decizie*. Această clasă conține toate acele probleme a căror soluție poate fi „Da” sau „Nu”. De exemplu, problema „Este graful conex?” este o problemă bidimensională (adică dacă între oricare două vârfuri ale grafului există un drum). Datele de intrare sunt vârfurile și muchiile grafului G (elemente care determină graful), iar întrebarea este următoarea: „Este graful G conex?”. De obicei, cele mai multe probleme de optimizare nu sunt probleme de decizie, dar se pot transforma ușor în astfel de probleme. De exemplu, una dintre versiunile problemei conisulului vorajor (*TSP – Traveling Salesman Problem*) are ca date de intrare graful G cu costurile muchiilor și un număr K .

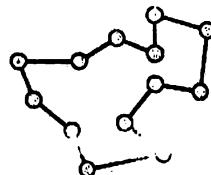
Întrebarea asociată este: „ G conține un tur de lungime mai mică sau egală cu K ?”. De obicei, o problemă de optimizare nu este mult mai greu de rezolvat decât problema de decizie corespunzătoare.



Clasa *P* se definește ca fiind mulțimea problemelor de decizie pentru care există un algoritm în timp polinomial de rezolvare, adică acele probleme care sunt considerate formal „ușoare” (accesibile).

Mai largă clasă de probleme *NP* conține clasa *P* și se referă la probleme cu algoritmi de rezolvare „nedeterministic-polinomiali”. Clasa *NP* conține acele probleme de decizie care au următoarea proprietate: pentru orice instanță „da” a problemei, rezultatul se poate verifica folosind un algoritm de tip polinomial. De exemplu, dacă considerăm problema: „Sunt grafurile G și G' izomorfe?”, „instanța da” a acestei probleme este o instanță în care grafurile G și G' sunt izomorfe. O demonstrație a acestei instanțe va fi un izomorfism (o mapare 1-1) f de la G la G' . Demonstrația răspunsului presupune, de fapt, verificarea faptului că f este într-adevăr un izomorfism de la G la G' , adică, pentru orice pereche de muchii $\{u, v\}$ din G , $\{f(u), f(v)\}$ este muchie în G' și G' nu are alte muchii în afară de cele verificate. Această demonstrație se poate realiza în timp polinomial relativ la dimensiunea grafurilor G și G' , deci această problemă aparține clasei *NP*. Un alt exemplu este și problema comisului voiajor, prezentată anterior în varianta de decizie, unde instanța este un graf G și un număr $K = 100$. Dacă răspunsul este „da”, atunci există o listă de muchii de lungime cel mult 100 care este un circuit; această listă acționează ca un „certificat” și se poate demonstra în timp polinomial că este o soluție validă, fapt suficient pentru a decide că răspunsul este „da”. Nu există simetrii între cele două variante „da” și „nu”. Dacă, pentru a ne convinge că răspunsul este „da”, este suficient să verificăm în timp polinomial o variantă de circuit, în cazul răspunsului „nu”, totul devine mult mai complicat (ar trebui să verificăm toate variantele posibile, ceea ce este extrem de costisitor, practic, imposibil). De fapt, prin schimbarea rolurilor jucate de „da” și „nu”, se obține o clasă de probleme numită *Co-NP* (de exemplu, pentru problema comisului voiajor: „toate circuitele posibile ale comisului voiajor în G au lungimea mai mare decât K ?“). Există și multe probleme ce par să se situeze în afara claselor *NP* și *Co-NP*, pentru că ele nu posedă o demonstrație în timp polinomial („certificat“). Un exemplu poate fi problema cu datele de intrare graful G și numerele K și L , cu întrebarea:

„Este cel mai lung circuit din graf G care conțineți vizitare, de lungime cel mult K , egal cu L ?“.



Probleme *NP*-complete



Până acum, nu s-a pășit un algoritm polinomial pentru rezolvarea problemei comisului voiajor. Pe de altă parte, nimic nu a fost capabil să demonstreze că un astfel de algoritm nu există. Deci se poate afirma că problema comisului voiajor, ca și multe alte probleme, este „intratabilă” (inaccesibilă). Se poate demonstra că varianta de decizie a problemei comisului voiajor, ca și multe alte probleme din clasa *NP*, sunt cele mai grele probleme *NP*, în sensul că, dacă există un algoritm în timp polinomial pentru una dintre aceste probleme, atunci există un algoritm polinomial

pentru **toate** problemele din **NP**. Observăm că aceasta este o afirmație foarte puternică, de vreme ce **NP** conține un număr imens de probleme care par să fie extrem de dificil de rezolvat atât teoretic, cât și practic! Problemele din **NP** cu această proprietate se numesc **NP-complete**.

Alte exemple de probleme NP-complete :

- problema determinării dacă două grafuri date sunt izomorfe;
- problema colorării grafurilor (*Graph Colouring Problem*): date fiind un graf G și un număr natural $k \geq 1$, există o k -colorare a lui G ? (k -colorare înseamnă posibilitatea de a se eticheta vârfurile grafului cu culorile $1, \dots, k$, astfel încât oricare două vârfuri vecine să fie colorate diferit);
- problema orarului (*Schedule Problem*): decizia existenței unui orar pentru un set de obiective, resurse și caracteristici ale unor activități;
- problema împachetării (*Bin Packing Problem*): date fiind o mulțime de obiecte cu dimensiunile lor și o mulțime de lázi, decizia dacă obiectele pot fi împachetate în lázi;
- problema partilionării (*Partition Problem*) unei mulțimi de numere întregi în două submulțimi, astfel încât suma numerelor din primul grup este egală cu suma numerelor din al doilea grup;
- problema acoperirii mulțimilor (*Set Cover Problem*): dată fiind o mulțime S , o colecție de submulțimi ale lui S și un număr natural k , să se decidă dacă există posibilitatea de a scrie pe S ca reuniune de k sau mai puține submulțimi din colecție;
- problema rucsacului (*Knapsack Problem*): date fiind un rucsac de dimensiune S , o mulțime de obiecte cu dimensiunile și greutățile lor și un număr întreg V , să se decidă dacă există o mulțime de obiecte pentru care suma dimensiunilor nu depășește S și suma greutăților este V sau mai mare.

Problema satisfiabilității (*SAT*)

Clasa **NP** și noțiunea de *probleme „complete” din **NP*** au fost prima dată introduse de Cook într-un articol din 1971. În articolul respectiv, el a demonstrat că o problemă particulară de decizie din logică, problema satisfiabilității (*SAT*), este **NP-completă**, dar și că orice altă problemă din **NP** poate fi coducibilă la trinu un caz special al problemei *SAT*. Odată rezolvată o problemă **NP-completă**, orice problemă **NP-completă** devine rezolvabilă, prin furnizarea unei transformări polinomiale între cele două. Putem afirma deci că *SAT* este una dintre cele mai importante probleme din teoria complexității.

Considerându-se o formulă logică în forma normală conjunctivă (CNF), să se decidă dacă există o asignare a variabilelor, astfel încât evaluarea formulei să alătura valoarea adevărat. O formulă logică este în formă normală conjunctivă (FNC) dacă are forma: $(y_{11} \vee y_{12} \vee \dots \vee y_{1m_1}) \wedge (y_{21} \vee y_{22} \vee \dots \vee y_{2m_2}) \wedge \dots \wedge (y_{m_1} \vee y_{m_2} \vee \dots \vee y_{mm_m})$, unde y_i sunt valori boolene din mulțimea de variabile $\{x_1, x_2, \dots, x_n, \bar{x}_1, \bar{x}_2, \dots, \bar{x}_n\}$.

Exemplu

Presupunem următoarea formulă în FNC:

$$F = (x_1 \vee \bar{x}_2) \wedge (x_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_3)$$

Problema este de a decide dacă există o asignare a variabilelor x_1, x_2, x_3 , astfel încât valoarea de adevăr a lui F corespunzătoare ei să fie 1. Observăm că în acest caz răspunsul este da, existând două astfel de asignări.

x_1	x_2	x_3	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	1	1	0

Clasa problemelor NP-hard

Clasa problemelor **NP-hard** este clasa problemelor cel puțin tot atât de grele ca orice problemă din **NP**. În particular, problemele de optimizare pentru care problema de decizie corespunzătoare este **NP-completă** (de exemplu, problema comisului voiajor) sunt **NP-hard**, deoarece versiunea de optimizare este cel puțin tot atât de dificilă ca și versiunea de decizie. Am văzut și mai sus că timpul real necesar găsirii unei soluții prin căutare, de la 1 la $n!$ de exemplu, este cel mai probabil deosebit de mare, și deci ocazional soluții optime este imposibilă. În ultimii ani, au apărut domenii ale informaticii care încearcă găsirea unor soluții „cât mai optime”: algoritmii evolutivi, inteligența artificială, sistemele fuzzy, toate fiind inspirate de natură.

Exerciții și probleme propuse

1. Scrieți un referat despre viața și contribuțiile lui Eratostene (Eratostenes) sau Euclid, folosind resurse de pe internet.

2. De unde provine cuvântul *algoritm*? Dați câteva definiții alternative.
3. Scrieți formal „algoritmul lui Euclid” și transformarea variabilelor pentru a afla cel mai mare divizor comun al numerelor 2322 și 654.
4. Scrieți formal algoritmul „sita lui Eratostene” și aplicați-l practic pentru a afla numerele prime mai mici decât 30.
5. Scrieți formal algoritmul de înmulțire a numerelor întregi.
6. Construiți formal algoritmul de determinare simultană a minimului și maximului dintr-o secvență de elemente. Care este complexitatea sa?
7. Scrieți algoritmii de căutare liniară, respectiv binară. Ce complexități au? Care este mai performant? Cum se explică acest lucru?
8. De ce putem afirma că o rețetă culinară este tot un algoritm?
9. Ce sunt *reprezentarea* și *transformarea* în cadrul ciclului de viață al unui program? Cum se codifică problema colorării hărților?
10. Care sunt proprietățile unui algoritm?
11. Ce înțelegem prin *proiectarea*, respectiv *analiza* algoritmului?
12. Ce se înțelege prin *model de calcul*? Dați exemple de modele de calcul. Descrieți proprietățile și avantajele modelului *RAM*.
13. Ce este *complexitatea* unui algoritm? Descrieți formal și grafic notațiile Θ , O și Ω .
14. Ce înțelegem prin *reducerea*, respectiv *optimalitatea* unui algoritm?
15. Cum se clasifică problemele relativ la complexitate? Dați exemple de probleme *NP-complete*.
16. Descrieți problema *SAT*. Dați și un exemplu sugestiv. De ce este ea considerată cea mai importantă problemă *NP-completă*?
17. Care este clasa problemelor *NP-hard*?
18. Care dintre următoarele afirmații sunt adevărate?

A) $n^2 \in O(n^3)$ B) $n^3 \in O(n^2)$ C) $2^{n+1} \in O(2^n)$ D) $(n+1)! \in O(n!)$
19. Determinați ordinul următorilor algoritmi:

pentru $i \leftarrow 1, n$ execută
pentru $j \leftarrow 1, 5$ execută
 {operație elementară}

pentru $i \leftarrow 1, n$ execută
pentru $j \leftarrow i, 0$ execută
 pentru $k \leftarrow 1, n$ execută
 {operăție elementară}

pentru $i \leftarrow 1, n$ execută
pentru $j \leftarrow 1, i + 1$ execută
 {operăție elementară}

pentru $i \leftarrow 1, n$ execută
pentru $j \leftarrow 1, i - 1$ execută
 pentru $k \leftarrow 1, n$ execută
 {operăție elementară}



Detaliu al unei fântâni publice din Zürich

Capitolul 2

Introducere în POO

- Concepte de bază
- Proprietăți ale programării orientate obiect
- Crearea și distrugerea obiectelor
- Elemente statice ale obiectelor
- Avantajele utilizării programării orientate obiect
- Probleme propuse

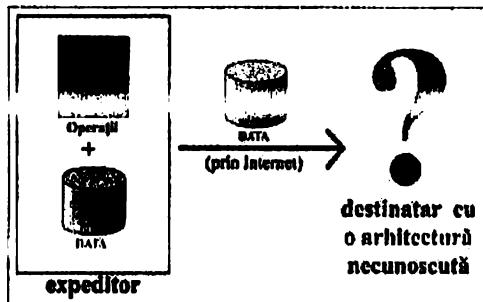
*When you discover a beautiful place like Positano
the first impulse you have is not to tell to anybody else your discovery.
You think: if I tell about this place to other people it will fill with tourists that will spoil it.*

John Steinbeck

Concepte de bază

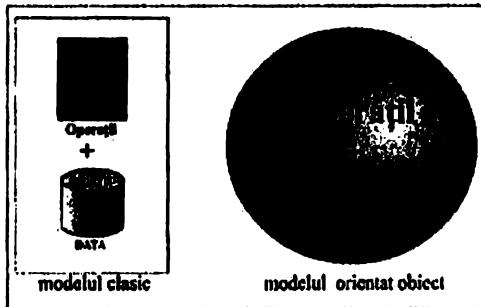
Conceptul de obiect este cheia de boltă în programarea orientată obiect (POO; în engl., *Object Oriented Programming – OOP*). În mare, obiectele soft în POO descriu obiecte din lumea reală, cum ar fi case, animale, clădiri; entități cu un caracter abstract, de genul formulelor matematice sau chiar evenimente ce pot avea loc (apăsarea unei taste, producerea unei explozii sau soldarea unui cont). Obiectele din lumea reală sunt caracterizate de o mulțime de atrbute. De exemplu, atrbutele unei cărți pot fi titlul, autorul, formatul, editura, culoarea copertăi și anul de apariție. Când obiecte din lumea reală, abstracții sau evenimente (acțiuni) sunt modelate cu ajutorul obiectelor soft, atrbutele acestor entități modelate sunt descrise cu ajutorul *variabilelor* din obiectul soft respectiv. Programarea structurată clasică (neorientată obiect sau procedurală) se bazează pe un model de programare ce separă operațiile (funcții, proceduri etc.) de datele asupra căror operațiile acționează. Rigiditatea acestui tip de programare reiese din controlul dificil al schimbărilor ce pot avea loc atât în structura datelor, cât și în cea a operațiilor.

Problema se complică și mai mult dacă se ia în calcul funcționarea programelor într-un mediu distribuit, în care arhitecturile hard și soft sunt foarte diverse. De cele mai multe ori, expeditorul unei informații (într-un mediu distribuit, cum ar fi internetul) nu știe aproape nimic despre arhitectura mașinii destinatar, excludându-se astfel posibilitatea existenței unui „translator” care să traducă datele destinatarului, astfel încât acesta să poată să le „înteleagă” (vezi figura de mai jos). Soluția la această problemă o oferă modelul de programare orientat obiect, în care datele și operațiile care le prelucrează sunt grupate în obiecte.



Transferul datelor prin internet

Deoarece datele și operațiile sunt plasate într-o singură entitate (numită obiect), comunicarea într-un mediu distribuit are loc mult mai ușor, practic, ea rezumându-se la transportul dintr-un loc în altul al obiectelor. Accesul destinatarului la date se va face prin operațiile furnizate de obiect. Ușurința acestui tip de comunicare la distanță este datorată existenței multor tehnologii sau standarde industriale ce permit comunicarea între obiecte aflate pe platforme diferite (exemplu: IIOP definit de Object Management Group).



După cum am precizat și mai înainte, datele sunt stocate în obiecte cu ajutorul variabilelor. Operațiile ce lucrează cu aceste date (funcții, proceduri etc.) poartă numele de *metode*. Un obiect poate fi văzut ca o mulțime de variabile și metode. În general, este recomandat că accesul la datele (variabilele) unui obiect să nu fie permis decât metodelor obiectului respectiv, realizându-se astfel o protecție sporită a datelor interne. Această proprietate a programării orientate obiect se numește *incapsulare*. *Exemplu:* La datele obiectului „carte” nu se poate ajunge decât prin intermediul metodelor dedicate (conținutul variabilei „pagini” nu se poate determina decât cu ajutorul metodei „nr_pagini()”). În C++ și Java, ascunderea datelor se face cu modificatorul de acționare “private”.

Un obiect este, de obicei, o componentă dintr-un program sau o aplicație, alături de alte obiecte, existența singulară a unui obiect nefiind prea folositoare. Tot secretul metodelor orientate pe obiecte constă în abilitatea de a face diferențe obiecte – fiecare descriind o anumită entitate – să interacționeze între ele și în urma producerii acestora să se obțină rezultatele scontate. *Exemplu:* Există o carte care poate să devină cărțe decât dacă există un alt obiect care să extragă informații din „carte”, apelând o metodă de tipul „citește()”.

Interacția sau comunicarea dintre obiecte este realizată prin *mesaje*. Dacă un obiect A dorește să execute o metodă a unui obiect B, atunci A îi trimite obiectului B un mesaj.

Uneori, obiectul destinatar (primitoarul mesajului) are nevoie de mai multe informații pentru a executa metoda cerută de expeditorul mesajului (solicitantul).

Exemplu: obiectul „student” trimite un mesaj obiectului „carte” prin care îl cere acestuia să apeleze metoda „deschide_carte()”. Această metodă a obiectului „carte” nu se poate executa, deoarece „student”-ul nu a precizat „cărții” pagina la care trebuie să se deschidă.

Aceste informații suplimentare, de care este nevoie pentru execuția unor metode, se numesc *parametri de intrare*. Pentru ca metoda „deschide_carte()” din obiectul „carte” să fie executată, trebuie ca expeditorul (în cazul nostru „student”-ul) să trimită în mesaj și valori pentru toți parametrii de intrare ai metodei ce se dorește să fie apelată („student”-ul trebuie să trimită în mesaj și valorile parametrului de intrare „pagina”, pentru a nu exista ambiguități vizavi de pagina la care va fi deschisă cartea).

O metodă poate avea zero, unul sau mai mulți parametri de intrare.

Unele metode, în urma execuției lor, pot furniza date numite *parametri de ieșire*. O metodă poate avea sau nu parametri de ieșire.

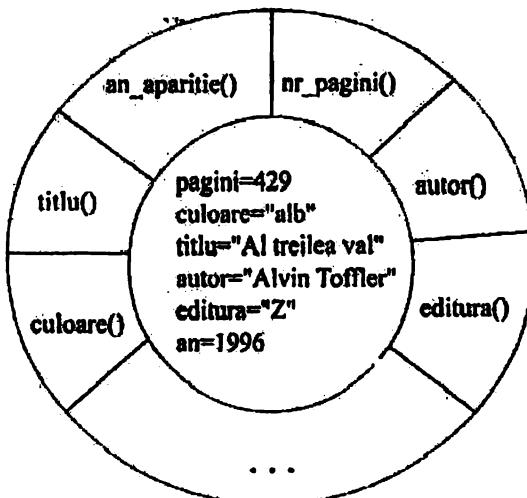
Exemple:

- Metoda „editura()” din obiectul „carte” nu are parametri de intrare și ca parametru de ieșire întoarce valoarea variabilei „editura”.
- Metoda „citește_pagina()” primește ca parametru de intrare numărul paginii și ca parametru de ieșire întoarce conținutul paginii al cărui număr a fost primit ca parametru de intrare.
- Metoda „deschide_carte” primește ca parametru de intrare numărul paginii și nu are parametri de ieșire, deoarece, în urma execuției metodei, nu este întoarsă nici o valoare (are loc doar operațiunea de deschidere a cărții).
- Metoda „închide_carte” nu are nevoie nici de parametrii de intrare, nici de cei de ieșire.

Un mesaj trimis unui obiect este compus din trei elemente:

1. numele obiectului care conține metoda;
2. numele metodei;
3. valori pentru toți parametrii de intrare ai metodei (dacă metoda are parametri de intrare).

Interacțiunea dintre obiecte este realizată prin apelarea metodelor sale, interacțiunea dintre obiecte are loc prin schimburi de mesaje. Obiectele nu trebuie să aparțină neapărat aceluiași proces și nici măcar nu este necesar să existe pe aceeași mașină pentru a putea trimite și primi mesaje.



O **clasă** este schema (planul) după care este creat un obiect. În interiorul unei clase, sunt precizate denumirile tuturor variabilelor și tipurile de valori pe care acestea le vor stoca (numeric, sir de caractere etc.), alături de toate operațiile (metodele) pe care un obiect, construit din respectiva clasă, le va conține.

Operația prin care este creat un obiect, după schema precizată de o clasă se numește **instantiere**. Obiectul creat după planul oferit de o clasă este o **instanță** a acelei clase. Nu se poate crea un obiect fără a se declara clasa (schema) după care acesta a fost instantiat.

În lumea reală, există foarte multe obiecte asemănătoare, descrise de aceeași mulțime de caracteristici. Cărțile, de exemplu, au toate titlu, autor, editură, an de apariție, format și gen și doar însoțite închise, cărți etc. Dacă se dorește crearea unei aplicații care să ia în considerare o mulțime de cărți, primul pas ar fi crearea unei clase „carte” în care sunt date denumirile variabilelor, tipul valorilor suportate de ele și metodele corespunzătoare. Pasul următor ar fi instanțierarea obiectelor de tip „carte”, fără în parte având valori specifice pentru variabilele sale (pentru autor, titlu, editură etc.).

Cu alte cuvinte, în programarea orientată obiect trebuie respectată următoarea ordine:

1. crearea claselor;

- precizarea denumirii variabilelor, a tipului de date suportat de fiecare variabilă în parte (tip standard: întreg, real, sir de caractere, etc. sau derivat o matrice, o altă clasă etc.), a domeniului de vizibilitate al variabilei (în general, este recomandată respectarea proprietății de încapsulare, prin

- folosirea de variabile având modifierul de acces setat pe *private* sau *protected*, astfel, accesul la variabile nefiind permis decât metodelor ce fac parte din aceeași clasă cu variabila) etc.;
- precizarea signaturii metodelor componente (a denumirii și a parametrilor de intrare, respectiv de ieșire), a corpurilor lor (codul care descrie comportamentul metodei), a domeniului de vizibilitate și a altor proprietăți mai mult sau mai puțin specifice unei anumite implementări a metodei orientate obiect.
- crearea obiectelor, prin specificarea clasei după a cărui model a fost creat obiectul respectiv și prin asignarea de valori variabilelor precizate în definiția clasei.

Doar în momentul instantierii unei clase (a nașterii unui obiect) variabilele definite de clasă sunt alocate în memorie și pot fi prelucrate de către metode.

Fiecare instanță (obiect) a unei clase are propria mulțime de valori, corespunzătoare variabilelor sale. De exemplu, fiecare carte are autorul și titlul proprii.

Toate instanțele unei clase suportă doar metodele precizate în definiția clasei, singura diferență între obiectele provenite din aceeași clasă constând în valorile diferite (dar de același tip) ce sunt stocate în variabile.

Proprietăți ale programării orientate obiect

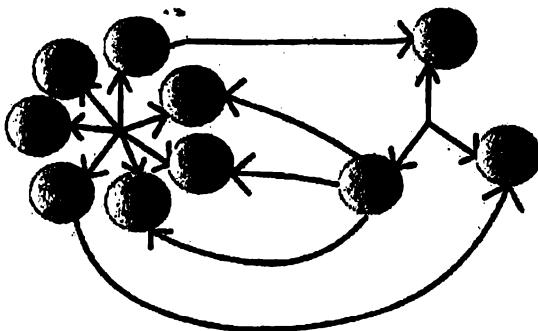
1. Încapsularea

Accesul la variabilele unui obiect se face doar prin apelarea metodelor sale, orice alt obiect neștiind nici măcar de existența variabilelor obiectului în cauză.

2. Moștenirea

Un alt concept important este moștenirea, care îi permite să se reutilizeze codul existent. Posibilitatea proiectării de noi clase folosind clase deja construite se numește *moștenire*. Dacă o clasă A moștenește o clasă B, atunci variabilele și metodele clasei B vor fi considerate ca aparținând și clasei A. Moștenirea permite crearea unei clase „de bază”, cu rolul de a fi o structură comună pentru clasele diferențiale, aceste proprietăți nu vor trebui precizate în fiecare clasă în parte.

Exemplu. Clasele „triunghi”, „patrulater” și „cerc” au structuri diferențiale, dar toate au o arie, un număr de laturi, perimetru etc. Pentru ca în cele trei clase să nu se repete aceleași elemente, variabile și metode, soluția optimă este proiectarea unei clase „figură geometrică”, în care să se regăsească trăsăturile comune și aceasta să fie moștenită de toate cele trei clase în cauză.



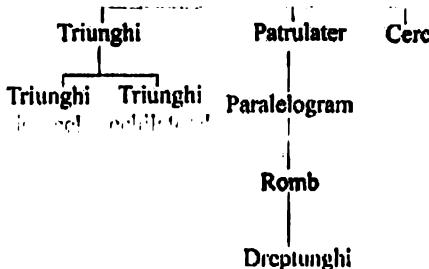
După cum se vede și din imaginea de mai sus, moștenirea este posibilă și pe mai multe niveluri. A nu se confunda moștenirea unei clase cu instanțierea ei. Numai o clasă poate moșteni proprietățile altor clase. În urma instanțierii unei clase, rezultă un obiect.

În urma moștenirii unei clase, clasa derivată (care moștenește proprietățile clasei de bază) nu copiează proprietățile clasei de bază, ci acestea îl sunt direct accesibile.

Datorită proprietății de moștenire, o clasă derivată nu numai că are acces la variabilele și metodele clasei moștenite, dar le și poate redefini. Datorită acestui fapt, o practică obișnuită în proiectarea orientată pe obiecte este declararea de metode cu același nume în clasele de bază, urmând ca acestor metode să li se ofere un corp în clasele derivate.

Exemplu. Clasa „figura geometrică” poate conține o metodă „aria()” care calculează aria figurii, dar o implementare a acesteia nu poate fi dată decât în clasele derivate (formula de calcul al ariei este diferită, în funcție de tipul figurii).

Figuri



Moștenirea ușurează munca proiectanților de sisteme informaticice prin modulitatea pe care o promovează; astfel, situații complexe din lumea reală pot fi mai ușor modelate.

3. Polimorfismul

Abilitatea unei metode cu un anumit nume de a avea comportări diferite în funcție de parametrii de intrare se numește *polimorfism*. Termenul *polymorph* provine din grecește, limbă în care înseamnă „a avea mai multe forme”. Nu trebuie să se confundă această proprietate cu cea de moștenire, cu ajutorul căreia se pot redefini metodele moștenite de la o altă clasă. Polimorfismul se referă la metode cu același nume și cu parametri de intrare și/sau ieșire diferiți. În ceea ce privește moștenirea, este vorba despre metode redefinite, dar care își păstrează aceeași signură (același nume și parametri de intrare/ieșire).

Polimorfismul are un rol foarte important în reutilizarea codului. Să presupunem că avem o clasă „operații aritmice” cu o metodă care primește ca parametri două numere oarecare (reale sau complexe). Dacă polimorfismul nu ar fi luat în calcul, atunci ar trebui implementate metode, fiecare cu o denumire diferită, pentru toate cele patru cazuri (număr real + număr real, număr real + număr complex, număr complex + număr real și număr complex + număr complex) și utilizatorul clasei ar trebui să țină minte toate cele patru nume de metode, pentru a face o adunare. Cu ajutorul polimorfismului, cele patru metode distincte pot avea același nume (în C++, de exemplu, supraîncărcarea operatorului +), iar programatorul care utilizează clasa „operații aritmice” nu trebuie să rețină decât o singură denumire, compilatorul alegând varianta corectă în funcție de parametrii de intrare furnizați.

Pe scurt, prin polimorfism, o metodă poate executa acțiuni diferite, în funcție de parametrii de intrare care îi sunt furnizați. Se realizează o interfață de comunicare a obiectului cu exteriorul mult mai unitară și mai flexibilă (metoda „adunare” are un singur nume, și nu patru), proiectanții putând să se concentreze mai bine asupra modului cum să folosească obiectele, fiind scutiți de detaliile de implementare.

În unele limbi de programare ce suportă metoda orientată pe obiecte, cum ar fi C++, este permisă supraîncărcarea operatorilor standard predefiniți. De exemplu, „+”, de obicei folosit pentru operația de adunare a două numere reale, poate fi extins să execute și alte tipuri de adunări.

Crearea și distrugerea obiectelor

Obiectele soft, ca și obiectele din lumea reală, au o viață limitată. Ele sunt create (înainte alocate resurse), folosite și distruse (resursele alocate sunt eliberate). După cum

am precizat anterior, la crearea unui obiect este absolut necesar să se preciseze modelul (clasa) după care este „turnat”, dar, pentru ca el să poată fi folosit, variabilele al căror tip și număr au fost precizate în definiția clasei, ar trebui să primească valori. Această operațiune de asignare de valori variabilelor obiectului în momentul nașterii sale sunt efectuate de niște metode speciale, numite *constructori*.

Într-o clasă, se pot defini mai mulți constructori, alegerea celui potrivit făcându-se în funcție de parametrii ceruți la instanțiere (la crearea unui obiect din clasa respectivă). Chiar dacă în definiția unei clase nu este implementat în mod explicit un constructor, sistemul generează un constructor implicit fără nici un parametru.

Existența constructorilor poate scuti programatorul de grija apelării în mod explicit a unor metode care să aloce resurse obiectului. Faptul că starea obiectului în momentul nașterii sale depinde de parametrii precizați (constructorul fiind ales în funcție de tipul și numărul acestora) conferă o mare flexibilitate codului, mai ales dacă se ia în considerare că metodele de tip constructor se pot moșteni.

La distrugerea obiectelor este necesară eliberarea resurselor alocate, acțiune care revine metodelor de tip *destructor*. *Exemplu:* un obiect, pe parcursul „vieții” sale, alocă dinamic spațiu de memorie; dacă în destructor spațiul de memorie respectiv nu este eliberat, atunci acesta nu va mai putea fi folosit de către alt proces. Acest tip de eroare este foarte periculos, putând duce la „sufocarea” (*memory leak*) sistemului din lipsă de resurse (pentru a evita situații de acest gen, creatorii limbajului de programare Java au eliminat posibilitatea alocării dinamice a memoriei prin pointeri și au introdus un mecanism automat de „colectare a gunoiului” – *garbage collection* –, prin care sunătatea resurselor ce au fost folosite de obiectele distruse).

O altă sarcină foarte importantă pe care destructorul trebuie să o îndeplinească este asigurarea că dispariția sa nu va crea nici o încurcătură. *Exemplu:* dacă se dorește stergerea unui obiect aflat într-o listă de obiecte, atunci este necesară „avertizarea” obiectelor vecine de la stânga și de la dreapta, pentru ca ele să stabilească „legătura

Elemente statice ale obiectelor

Pe lângă variabilele și metodele declarate în mod „normal”, există și declarările lor *static*. Dacă o variabilă este declarată statică în definiția unei clase, atunci compilatorul va crea doar o singură copie a ei, indiferent de numărul obiectelor instanțiate din clasa respectivă. Toate aceste obiecte vor face referire practic la singură variabilă, deci valoarea stocată de ea va fi același pentru toate obiectele, spre deosebire de „elementele normale”, pentru care există câte o copie aferentă fiecărui obiect. *Exemplu:* variabila „cont” în clasa *Object* din programul problemei 1 (problemă fractionară a rucsacului) din capitolul *Greedy*.

Restricții:

- O metodă statică poate folosi doar elemente statice din clasa în cauză.
- Nu se pot supraîncărca metode statice cu metode non-statice și nici invers.

Elementele statice sunt folosite mai ales pentru controlul la nivel de clasă al programelor. Cu ajutorul lor se poate controla comportamentul tuturor obiectelor instanțiate dintr-o anumită clasă.

Avantajele utilizării programării orientate obiect

Ușurința proiectării și reutilizării codului

Odată ce este testată corectitudinea funcționării unor obiecte dintr-o aplicație, acestea vor putea fi folosite fără nici o problemă și în altă aplicație. Acest avantaj poate fi valorificat prin constituirea de biblioteci de obiecte. În ceea ce privește proiectarea, se facilitează descompunerea problemelor complexe în subprobleme simple, care pot fi ușor modelate cu ajutorul obiectelor (variabilele vor descrie proprietățile obiectelor modelate și metodele acțiunilor lor).

Abstractizare

Proiectanții pot obține o imagine de ansamblu, urmărind comportarea obiectelor și interacțiunile dintre ele, detaliile fiind înglobate în compoziția obiectelor.

Siguranța datelor

Abilitatea obiectelor de a se comporta ca niște „cutii negre”, de a putea fi folosite fără a se cunoaște compoziția lor, asigură confidențialitatea datelor utilizate și micșorează frecvența erorilor și efectul erorilor însele din programare.

Probleme propuse

1. Ce este abstractizarea datelor? Selecțiați trei exemple din carteia de lață.
2. Care este diferența dintre un obiect și o clasă? Se poate crea un obiect fără a crea clasa după care este instanțiat?
3. Care este diferența dintre o variabilă locală și o variabilă membru a unei clase?
4. Care este diferența dintre un câmp static și unul nestatic?
5. Cum se pot ascunde datele în C++?
6. Ce este polimorfismul? Găsiți exemple de polimorfism în programele cărții de față.

7. Care este mecanismul de creare și distrugere a obiectelor în C++? La ce poate conduce păstrarea obiectelor care nu mai sunt necesare dintr-un anumit punct al aplicației?
8. Ce este moștenirea? Găsiți trei exemple sugestive și implementați programe simple pentru a ilustra conceptul.
9. Enumerați avantajele utilizării programării orientate obiect.
10. Explicați conceptul de *încapsulare*. Dați trei exemple.



Capitolul 3

Introducere în STL

10 probleme complet rezolvate, 16 probleme propuse

- **Istoric, concepte de bază**
- **Programarea generică**
- **Containere**
- **Iteratori**
- **Algoritmi**
- **Functori**
- **Încă șapte probleme rezolvate**
- **Întrebări, exerciții, probleme propuse**

*Possibly one of the most boring terms in history,
for one of the most exciting tools in history.*

Scott Field

Istoric, concepte de bază

Standard Template Library (STL) este o parte foarte importantă a limbajului C++, devenită standard al limbajului încă din 1999. Ea furnizează diverse implementări ale unor structuri și algoritmi de utilizat, cum ar fi containere, iteratori, algoritmi, functori și alocatori. Este o bibliotecă generică, aceasta însemnând că aproape toate componentele sale sunt parametrizate: fiecare componentă este în STL un şablon (tip generic, *template*). Este indicat să fie utilizată biblioteca STL oricând este posibil, deoarece este foarte practică, eficientă și robustă.

Programarea generică

Programarea generică este o paradigmă de programare ce permite implementarea unor clase și metode, astfel încât să poată fi folosite cu diverse tipuri de date. Diferite limbiage de programare suportă conceptul de tipuri generice (şabloane, *template-uri*, sau polimorfie parametrizată), dar trebuie să satisfacă o serie de condiții. Astfel de limbiage poartă numele de *limbiage generice*. În header-ul de algoritmi al STL se găsesc numeroși algoritmi de bază, des utilizati în implementarea aplicațiilor, care sunt cât mai independent posibil de tipurile de date cu care lucrau.

Ca exemplu, scriem un program care implementează o funcție de interschimbare a două elemente. Deoarece tipul este generic, vom putea folosi aceasta funcție pentru orice tipuri de variabile. În programul principal se testează pentru siruri de caractere (`std::string`), tipul `float` și vectori.

Problema 1. Scrieți un program care implementează o funcție generică pentru interșimbarea a două elemente și o testează pentru tipuri diferite de argumente.

Program 1

```
#include <string>
#include <vector>
#include <iostream>

using namespace std;

template <class T> void interschimba(T &x, T &y){
    T aux;
    aux = x;
    x = y;
    y = aux;
}

int main(){

    string s1 = " ziua!";
    string s2 = "Buna";
    interschimba(s1, s2);
    cout << s1 << s2 << endl
        << endl;

    float f1, f2;
    cout << "f1 = "; cin >> f1;
    cout << "f2 = "; cin >> f2;
    cout << "-----" << endl;
    interschimba(f1, f2);
    cout << "f1 = " << f1 << endl
        << "f2 = " << f2 << endl
        << endl;

    vector<int> v1;
    vector<int> v2;
    int i;
    for (i=0; i<10; i++) v1.push_back(i);
    for (i=100; i>50; i-=10) v2.push_back(i);
    interschimba(v1, v2);

    cout << "v1: ";
    for (i=0; i<(int)v1.size(); i++) cout << v1[i] << " ";
    cout << endl << "v2: ";
    for (i=0; i<(int)v2.size(); i++) cout << v2[i] << " ";
}
```

Efectul programului

```
Buna ziua!
x1 = 4.53
x2 = 2.09
-----
f1 = 2.09
f2 = 4.53

y1: 100 90 80 70 60
y2: 0 1 2 3 4 5 6 7 8 9
```

- Observăm astfel că perechile de variabile au fost inversate folosind aceeași funcție `interschimba()`, dar cu parametri de tipuri diferite (specializare explicită). Acest lucru este posibil datorită tipului generic `T`, folosit în implementarea metodei.

Containere

Containerele STL (colecții) implementează multe modalități de a stoca datele, împreună cu o serie de metode uzuale pentru modificarea și exploatarea acestora. Ele pot stoca orice tip de elemente și cele mai multe dintre ele sunt generalizate ca `template-uri`. Foarte des utilizate sunt containerele `vector`, `string` și `map`.

Containerele sevențiale conțin sevențe de lungime variabilă de elemente de același tip `T`:

- `<vector>`: tablou dinamic cu elemente de un tip dat `T`, acces rapid la orice element;
- `<deque>`: de asemenea, un tablou dinamic de elemente, care aduce în plus inserarea și elrimarea rapida ale unui element de la începutul, respectiv sfârșitul sevenței;
- `<list>`: este implementată ca o listă dublu înăntălită, inserarea și eliminarea unui element la ambele capete ale sevenței se execută foarte rapid.

Containerele asociative conțin perechi cheie/valoare, în care accesul la valori se face prin intermediul cheilor. Elementele sunt sortate după cheie și sunt implementate ca arbori binari echilibrați:

- `<map>`: furnizează accesul la valori folosind orice tip de cheie. Elementele sunt de tipul `pair<const Key, T>`, unde primul element este cheia sortată, iar al doilea este valoarea asociată ei. Pentru fiecare cheie există cel mult o valoare. Tipul (clasa) pentru cheie trebuie să suporte operația de comparație „mai mic” pentru a putea executa operația de sortare;

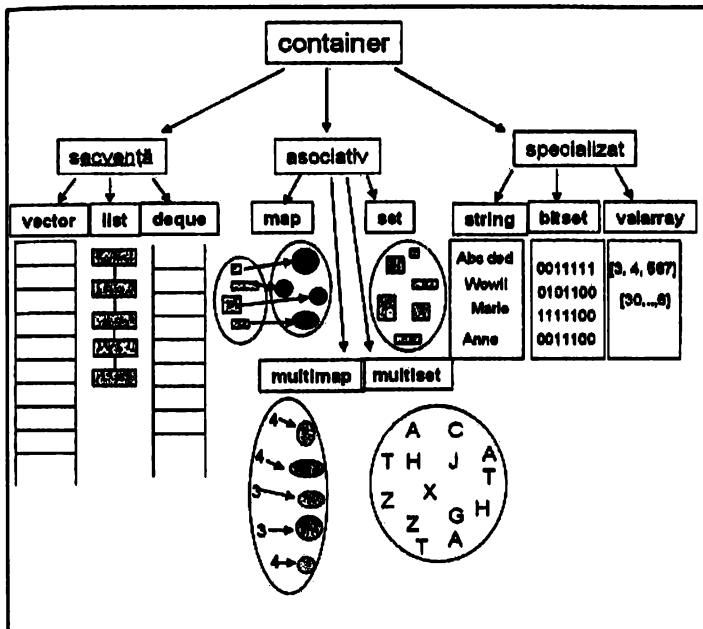
- ***multimap***: este același lucru ca și *map*, numai că este permisă apariția repetată a același chei. Declarația clasei se găsește în `<map>`;
- ***<set>***: furnizează o mulțime ordonată de elemente (fiecare element apare exact o dată), nu implementează (așa cum ar fi de așteptat) și operațiile uzuale pe mulțimi (reuniune, intersecție etc.);
- ***multiset***: mulțimul ordonată de elemente (un element poate să apară repetat). Declarația clasei se află în `<set>`;

Containerele adaptor sunt construite pe baza altor containere și sunt folosite doar pentru a forța regulile de acces. Tocmai din cauza acestor restricții de acces, containerele adaptor nu suportă iteratori:

- ***<stack>***: structură de tip stivă, oferă acces *LIFO* (*Last In First Out* – ultimul adăugat este primul extras);
- ***<queue>***: structură de tip coadă, oferă acces *FIFO* (*First In First Out* – primul adăugat este primul extras);
- ***priority_queue***: coadă în care elementele au priorități, care determină și ordinea de acces la ele. Declarația sa se găsește în `<queue>`.

Containerele specializate conțin elemente specializate; de exemplu, tipuri de date specifice, rutine utilitare, implementări limitate (dar eficiente). *Stroustrup* numește aceste containere „aproape containere”, datorită faptului că fiecare dintre ele are o serie de limitări comparativ cu containerele standard:

- ***<string>***: oferă multe funcții utile și eficiente pentru lucrul cu siruri de caractere;
- ***<bitset>***: un *bitset*<N> este un câmp cu N biți. El se diferențiază de *vector<bool>* prin faptul că are o lungime fixă. Pentru mulțimi mici de biți, C++ oferă posibilitatea folosirii operatorilor pe biți pentru numere întregi (& – și, | – sau, ^ – sau exclusiv, ~ – clăbucare spre stânga, >> – deplasare spre dreapta, ~ – negație). Clasa *bitset*<N> generalizează această abordare și oferă abordarea comodă a mulțimilor de N biți indexate de la 0 la N – 1, unde N trebuie să fie cunoscut la compilare;
- ***<valarray>***: este un vector optimizat pentru operații numerice, dar nu are o comportare de containere standard.



Containere (colecții de date) uzuale în STL

Timpul necesar pentru operații uzuale asupra containerelor secvențiale este redat în tabelul de mai jos.

Operatie	vector	dequeue	list
Acces la primul element	constant	constant	constant
Acces la ultimul element	constant	constant	constant
Acces la un element aleator	constant	constant	liniar
Adăugare/stergere la început	liniar	constant	constant
Adăugare/stergere la sfârșit	constant	constant	constant
Adăugare/stergere aleatorie	liniar	liniar	constant

Unele elemente pot fi accesate direct:

Sintaxă	Semantică
front()	primul element
back()	ultimul element
[]	acces fără verificare la un index dat (nu și pentru liste)
at()	acces verificat la un index dat (nu și pentru liste)

Majoritatea containerelor oferă posibilitatea de a executa operații eficiente cu ultimul element al acestora. Pentru list și dequeue, se oferă operații și pentru primele elemente. Operații de tip stivă și coadă asupra containerelor:

Sintaxă	Semantică
<code>push_back(x)</code>	adăugarea unui element la sfârșit
<code>pop_back()</code>	stergerea ultimului element
<code>push_front(x)</code>	adăugarea unui element la început (numai pentru list și dequeue)
<code>pop_front()</code>	stergerea primului element (numai pentru list și dequeue)

Containerele oferă și operații specifice listelor, cum ar fi:

Sintaxă	Semantică
<code>insert(p, x)</code>	adăugarea elementului <i>x</i> înaintea lui <i>p</i>
<code>insert(p, n, x)</code>	adăugarea a <i>n</i> copii ale lui <i>x</i> înaintea lui <i>p</i>
<code>insert(p, pb, pe)</code>	adăugarea înaintea lui <i>p</i> a elementelor dintre <i>pb</i> și <i>pe</i>
<code>erase(p)</code>	sterge elementul de la poziția <i>p</i>
<code>erase(pb, pe)</code>	sterge elementele dintre <i>pb</i> și <i>pe</i>
<code>clear()</code>	stergerea tuturor elementelor

Toate containerele oferă alte operații uzuale:

Sintaxă	Semantică
<code>size()</code>	returnează numărul elementelor
<code>empty()</code>	testează dacă are elemente
<code>max_size()</code>	dimensiunea celui mai mare container posibil
<code>capacity()</code>	spațiul rezervat (numai pentru vector)
<code>clear()</code>	sterge toate elementele
<code>resize()</code>	schimbarea numărului de elemente (numai vector, deque, list)
<code>swap()</code>	interschimbarea elementelor a două containere
<code>!=</code>	verifică dacă două containere au conținuturi diferite
<code><</code>	compararea ordinii lexicografice

Constructorii și destrutorii pentru containere:

Sintaxă	Semantică
<code>container()</code>	container vid
<code>container(n)</code>	container cu n elemente <i>default</i> (exclus pentru containerele asociative)
<code>container(n, x)</code>	n copii ale lui x (exclus pentru containerele asociative)
<code>container(pb, pe)</code>	initializarea cu elementele dintre pb și pe
<code>container(c)</code>	constructor de copiere; copie elementele din c
<code>-container()</code>	destructor; sterge containerul și toate elementele sale

Operații de alocare:

Sintaxă	Semantică
<code>operator=(c)</code>	asignare de copiere; copie elementele din containerul c
<code>assign(n, x)</code>	asignarea a n copii ale lui x (exclus pentru containerele asociative)
<code>assign(pb, pe)</code>	asignează elementele dintre pb și pe

Containerele asociative oferă accesul la elemente cu ajutorul cheii:

Sintaxă	Semantică
<code>operator[](k);</code>	accesul la elementul cu cheia k (pentru containerele cu cheie unică)
<code>find(k)</code>	căutarea elementului cu cheia k
<code>lower_bound(k)</code>	căutarea primului element cu cheia k
<code>upper_bound(k)</code>	căutarea primului element cu cheie mai mare decât k
<code>equal_range(k)</code>	găsește <code>lower_bound()</code> și <code>upper_bound()</code> pentru k

Iteratori

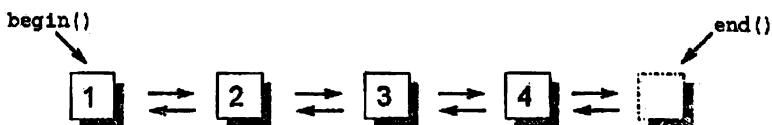
Iteratorii sunt folosiți pentru a naviga între elementele unui container, fără a fi necesar să știm tipul elementelor acestuia. El este asociat la declarare unui anumit tip de element și poate fi folosit pentru a accesa elementele container-ului în mod similar altul.

Iteratorii sunt abstractizări generalizate ale pointerilor, văzuți deci ca niște „cutii negre”, folosite de cele mai multe ori în practică pentru a aplica algoritmii asupra containерelor. Dacă se utilizează un telefon pentru a suna pe cineva în altă țară, de exemplu, nu este necesară întreaga informație despre sistemul telefonului-destinație. Este necesar însă ca acesta să suporte o serie de operații de bază, cum ar fi: selectarea numărului, soneria, raportarea stării, închiderea conexiunii la așezarea receptorului,

care sunt necesare pentru stabilirea comunicării. În aceeași manieră, dacă o clasă suportă un minimum de tipuri de iteratori pentru un anumit algoritm, atunci algoritmul va funcționa pentru respectivul container.

Condiția care se folosește de obicei pentru terminarea unei bucle este puțin diferită de cea normală. În mod ușor, se folosește de multe ori condiția „mai mic sau egal”. În cazul iteratorilor, se pornește, de obicei, de la `begin()` și condiția de oprire este înainte de egalitatea cu `end()`. Este important de știut că `end()` returnează un iterator care pointează la un element imediat după ultimul. Acesta nu se poate dereferenția, dar se folosește pentru teste de egalitate cu alți iteratori de același tip. Operatorul de preincrementare `++` poate conduce uneori la performanțe superioare operatorului de postincrementare, deoarece nu creează o copie temporară a valorii precedente.

Perechea `begin()/end()` furnizează elementele containerului în ordinea directă folosind `iterator` (elementul 0, elementul 1, elementul 2 și.a.m.d.). Perechea `rbegin()/rend()` furnizează elementele containerului în ordine inversă folosind un `reverse_iterator` (elementul $n-1$, elementul $n-2$, elementul $n-3$ și.a.m.d.). O secvență parcursă cu `iterator` poate fi astfel reprezentată:



Cu `reverse_iterator` se poate reprezenta astfel:



Programul pentru problema următoare este sugestiv privind utilizarea celor două variante de iteratori.

Problema 2. Scrieți un program care citește de la tastatură un vector de numere reale și apoi îl afișează folosind `iterator` și `reverse_iterator`.

Program 2

```

#include <iostream>
#include <vector>

using namespace std;
int main(){
    vector<short> v;
    short n, t;
    cout << "n = ";
    cin >> n;
    cout << "Elementele: ";
    for (short i=0; i<n; i++){
        cin >> t;
        v.push_back(t);
    }
    vector<short>::iterator it;
    cout << "\nParcursere directa: ";
    for(it=v.begin(); it<v.end(); ++it){
        cout << *it << " ";
    }

    vector<short>::reverse_iterator rIt;
    cout << "\nParcursere inversa: ";
    for(rIt=v.rbegin(); rIt<v.rend(); ++rIt){
        cout << *rIt << " ";
    }
}

```

Efectul programului:

n = 5

Elementele: 34 -23 4 67 1

Parcursere directa: 34 -23 4 67 1

Parcursere inversa: 1 67 4 -23 34

Algoritmii din *header-ul <algorithms>*, cum ar fi, de exemplu, *sort* sau *random_shuffle*, nu cunosc tipul containerelor, dar folosesc anumiți iteratori, ce trebuie să fie suportați de acestea. Iteratorii pot fi priviți și ca o interfață dintre containere și algoritmi. Există cinci categorii de iteratori:

- iterator cu acces uniform: *Random_Access_Iterator*;
- iterator bidirecțional: *Bidirectional_Iterator*;
- iterator de înaintare: *Forward_Iterator*;

- iterator de intrare: *Input_Iterator*;
- iterator de ieșire: *Output_Iterator*.

Pentru toate variantele de iteratori, există și variante *const* (*const_iterator*, *const_reverse_iterator*), care se referă la iteratori ce nu modifică starea elementelor referențiate. Aceștia sunt recomandați în cazul în care nu este necesară modificarea conținutului elementelor.

Algoritmi

Algoritmii *STL* sunt funcții generice C++, care execută operații uzuale asupra containerelor. Pentru a putea fi capabili să opereze cu diferite tipuri de containere, ei nu primesc ca parametri containerele, ci iteratori care specifică o parte a lor. Unii algoritmi au nevoie doar de suport pentru intrare (*Input_Iterator*), pe când alții au nevoie de acces uniform (*Random_Access_Iterator*, de exemplu, pentru *sort()*).

Utilizarea algoritmilor pare la prima vedere dificilă, dar, odată folosiți, devine foarte practică. Mulți dintre aceștia au interfețe foarte asemănătoare, ce ușurează înțelegerea lor. În biblioteca `<algorithm>` se găsesc 60 de algoritmi care includ operații de sortare (*sort*, *merge*, *min*, *max* etc.), căutare (*find*, *count*, *equal* etc.), operații de mutare (*transform*, *replace*, *fill*, *rotate*, *shuffle* etc.) și operații numerice generalizate (*accumulate*, *adjacent_difference* etc.). Ei se găsesc în spațiul de nume *std* și sunt declarați în `<algorithm>`.

Operațiile care nu modifică secvențele sunt cele care extrag informații despre un container; de exemplu, poziția unui element în container sau numărul de elemente:

Sintaxă	Semantică
<code>for_each()</code>	pentru fiecare element din intervalul dat execută
<code>find()</code>	prima apariție a unei valori
<code>find_if()</code>	primul element care satisfacă un predicat
<code>find_first_of()</code>	element al unei secvențe în altă secvență
<code>adjacent_find()</code>	găsirea unei perechi de elemente
<code>count()</code>	numărul de elemente cu o anumită valoare
<code>count_if()</code>	numărul de elemente care satisfac un predicat
<code>mismatch()</code>	primele elemente diferite a două secvențe
<code>equal()</code>	testează dacă două secvențe conțin aceleși valori
<code>search()</code>	prima apariție a unei subsecvențe
<code>find_end()</code>	ultima apariție a unui element
<code>search_n()</code>	prima subsecvență cu <i>n</i> elemente egale

Algoritmii care modifică secvențele:

Sintaxă	Semantică
<code>transform()</code>	pentru fiecare element din intervalul dat execută operație
<code>copy()</code>	copierea tuturor elementelor începând cu primul
<code>copy_backwards()</code>	copierea tuturor elementelor începând cu ultimul
<code>swap()</code>	interschimbarea a două elemente
<code>iter_swap()</code>	interschimbarea a două elemente indicate de doi iteratori
<code>swap_ranges()</code>	interschimbarea tuturor elementelor a două secvențe
<code>replace()</code>	înlocuirea unor elemente cu o altă valoare
<code>replace_if()</code>	înlocuirea unor elemente care satisfac un predicat
<code>replace_copy()</code>	copierea cu înlocuirea unei valori date
<code>replace_copy_if()</code>	copierea cu înlocuirea unor elemente care satisfac un predicat
<code>fill()</code>	înlocuirea tuturor elementelor cu o valoare dată
<code>fill_n()</code>	înlocuirea a n elemente cu o valoare dată
<code>generate()</code>	înlocuirea tuturor elementelor cu rezultatul unei operații
<code>generate_n()</code>	înlocuirea a n elemente cu rezultatul unei operații
<code>remove()</code>	stergerea elementelor cu o anumită valoare
<code>remove_if()</code>	stergerea elementelor care satisfac un predicat
<code>remove_copy()</code>	copierea elementelor cu eliminarea unei valori
<code>remove_copy_if()</code>	copierea cu eliminarea elementelor ce satisfac un predicat
<code>unique()</code>	eliminarea duplicatelor consecutive
<code>unique_copy()</code>	copierea elementelor cu eliminarea duplicatelor consecutive
<code>reverse()</code>	inversarea elementelor secvenței
<code>reverse_copy()</code>	copierea elementelor în ordine inversă
<code>rotate()</code>	rotirea elementelor
<code>rotate_copy()</code>	copierea elementelor cu rotirea lor
<code>random_shuffle()</code>	amestecarea elementelor

STL oferă și foarte mulți algoritmi pentru sortarea, căutarea și manipularea secvenților sortate.

Sintaxă	Semantică
<code>sort()</code>	sortarea secvenței
<code>stable_sort()</code>	sortarea (aceleși elemente își păstrează ordinea)
<code>partial_sort()</code>	sortarea primici părți a secvenței
<code>partial_sort_copy()</code>	copierea și apoi sortarea primei părți

<code>nth_element()</code>	poziționarea celui de-al n -lea element pe poziția finală în secvența sortată
<code>lower_bound()</code>	găsirea primului element în secvența sortată
<code>upper_bound()</code>	ultimul element în secvența sortată
<code>equal_range()</code>	găsirea unei subsecvențe cu o anumită valoare
<code>binary_search()</code>	găsirea unei valori într-o secvență sortată
<code>merge()</code>	combinarea a două secvențe sortate
<code>inplace_merge()</code>	combinarea a două secvențe sortate consecutive
<code>partition()</code>	plasarea în față a elementelor care satisfac un predicat
<code>stable_partition()</code>	plasarea în față a elementelor care satisfac un predicat, cu păstrarea ordinii relative

Algoritmii pentru mulțimi:

Sintaxă	Semantică
<code>includes()</code>	verifică inclusiunea unei secvențe în alta
<code>set_union()</code>	reuniunea a două secvențe
<code>set_intersection()</code>	intersectia a două secvențe
<code>set_difference()</code>	mulțime sortată cu elementele care sunt în prima secvență și nu sunt în a doua
<code>set_symmetric_difference()</code>	mulțime sortată cu elementele care se află exact în una dintre cele două mulțimi date

Operațiile `heap` păstrează o secvență într-o stare astfel încât rămâne simplu de sortat, atunci când este nevoie:

Sintaxă	Semantică
<code>make_heap()</code>	pregătirea unei secvențe pentru manipularea ca <code>heap</code>
<code>push_heap()</code>	adăugarea unui element în <code>heap</code>
<code>pop_heap()</code>	eliminarea unui element din <code>heap</code>
<code>sort_heap()</code>	sortarea <code>heap</code> -ului

Algoritmii pentru selecție: cea mai mare sau cea mai mică secvență

Sintaxă	Semantică
<code>min()</code>	returnează elementul mai mic dintre două valori
<code>max()</code>	returnează cea mai mare dintre două valori
<code>min_element()</code>	returnează cel mai mic element dintr-o secvență

<code>max_element()</code>	returnează cel mai mare element dintr-o secvență
<code>lexicographical_compare()</code>	compararea lexicografică a două secvențe

STL oferă următoarele metode pentru operațiile cu permutări:

Sintaxă	Semantică
<code>next_permutation()</code>	următoarea permutare în ordine lexicografică
<code>prev_permutation()</code>	permutarea precedentă în ordine lexicografică

Functori

Un obiect de tip funcție (*functor*) este un obiect care poate fi apelat ca și funcție. O funcție obișnuită este un astfel de obiect; la fel este un obiect ce are definit sau supraîncărcat operatorul de apel de funcție (*operator()*). Aceste obiecte de tip funcție au o importanță deosebită pentru *STL*.

Conceptele de bază pentru obiecte de tip funcție sunt *generator*, *funcție unară* și *funcție binară*: acestea descriu, respectiv, dacă obiectele pot fi apelate ca $f()$, $f(x)$ sau $f(x, y)$. Această listă poate fi extinsă la *funcție ternară* etc., dar, în practică, toți algoritmii din *STL* folosesc functori cu cel mult două argumente. Toate conceptele de obiecte de tip funcție folosite în *STL* sunt rafinări ale acestora trei.

Obiectele de tip funcție care returnează *bool* sunt un caz special. O funcție unară ce returnează *bool* se numește *predicat (unar)*, o funcție binară ce returnează *bool* se numește *predicat binar*.

Problema 3. Scrieți un program care găsește poziția primului element prim într-un vector de numere naturale, folosind două variante: căutarea secvențială în vector și metoda *find_if* din *header-ul algorithm* cu al treilea parametru *predicat binar* (funcție de testare a unui bit).

Program 3

```
#include <vector>
#include <iostream>
#include <algorithm>

using namespace std;

bool prim(unsigned n){
    if(n>2 && n%2==0) return false;
    for(unsigned i=3; i*i<=n; i+=2)
        if(n%i==0) return false;
```

```

    return true;
}

int main(){
    vector<unsigned> v;
    int n, t;
    cout << "n = "; cin >> n;
    cout << "Elementele: ";
    while(n--){
        std::cin >> t;
        v.push_back(t);
    }

    // prima variantă: căutare iterativă în vector
    vector<unsigned>::iterator it;
    bool found = false;
    for(it=v.begin(); !found && it!=v.end(); it++)
        if(prime(*it)) found = true;
    if(found){
        cout << "Primul element prim la pozitia "
            << it-v.begin();
    }else{
        cout << "Tabloul nu contine elemente prime!";
    }

    // a doua variantă: prim este predicat binar pentru algoritmul find_if
    it = find_if(v.begin(), v.end(), prime);
    if(it!=v.end()){
        cout << "\nPrimul element prim la pozitia "
            << it - v.begin() + 1;
    }else{
        cout << "\nTabloul nu contine elemente prime!";
    }
}

```

Efectul programului:

```

n = 5
Elementele: 45 6 78 37 56
Primul element prim la pozitia 4
Primul element prim la pozitia 4

```

Prima variantă folosește un iterator, care se deplasează de la stânga spre dreapta în vector până când elementul curent este prim sau se ajunge la sfârșitul vectorului. A două variantă utilizează algoritmul `find_if` din header-ul `<algorithm>`; returnează primul

lement pentru care predicatul binar este satisfăcut; dacă nu există un astfel de predicat, va returna *v.end()*.

Încă șapte probleme rezolvate

Problema 4. Scrieți un program care alege aleator o variantă pentru LOTO 6/49.

Program

```
include <vector>
include <iostream>
include <algorithm>
include <iterator>

const short NMAX = 49;
const short M = 6;

int main(){
    std::vector<short> v;
    for(short i=0; i<NMAX; i++)
        v.push_back(i+1);
    random_shuffle(v.begin(), v.end());
    copy(v.begin(), v.begin()+M,
        std::ostream_iterator<short> (std::cout, " "));
    return 0;
}
```

Efectul programului

```
12 1 9 44 14 27
```

Vectorul *v* este completat dinamic cu numerele 1, 2, ..., *NMAX*, apoi este amestecat folosind metoda *random_shuffle()*, după care se vor afișa primele *M* elemente ale sale (presupunem că *M* ≤ *NMAX*). Pentru scrierea în fluxul de ieșire (pe ecran – *std::cout*), folosim metoda *copy()*. același din program este echivalent și cu:

```
for(short i=0; i<M; i++)
    std::cout << v[i] << " ";
```

Că exercițiu, modificați programul astfel încât afișarea să se facă folosind iteratori (variantele *iterator* și *reverse_iterator*) și folosind metoda *at()* (acces verificat) în locul metodei *copy*.

Problema 5. Scrieți un program care afișează parcurgerea DFS (depth first search) a unui graf dat prin matricea de adiacență. Exemplu:

graph.in	dfs.out	
<pre>8 1 0 1 1 1 0 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 0 0 0 1 1 0 0 0 1 0 0 0 0 1 0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 0 0 1 0 0</pre>	<pre>1 2 5 3 7 6 4 8</pre>	

Acest algoritm traversează graful în adâncime. Începând cu vârful inițial, se va vizita primul vecin al acestuia încă neprocesat, după aceea primul vecin neprocesat al acestuia și.a.m.d. Atunci când un nod nu mai are vecini neprocesați, se va trece la predecesorul acestuia, pentru care se caută vecinii neprocesați. Algoritmul folosește o stivă (engl. stack):

ALGORITM_DFS(Graf G, Vârf y)

Inițializează stiva S cu {y}

While (S NOT Null) Do

k \leftarrow S.top() // elemenul din vârful lui S, fără stergerea acestuia

Închideză și proceseză k

j \leftarrow primul vecin nevizitat al lui k

If (j există) Then

Marchează și procesează j

S.push(j)

End_If

Else S.pop() End_Else // șterge elementul din vârful lui S

End_While

ALGORITM_DFS(Graf G, Vârf y)

Modelăm algoritmul de mai sus folosind containerul adaptor std::stack.

Program 5

```
#include <stack>
#include <fstream>
#include <vector>
#include <algorithm>
```

```

int main(){
    std::ifstream in("graph.in");
    std::ofstream out("dfs.out");
    short n, k, a[100][100];
    std::stack<short> st;
    std::vector<short> v;
    in >> n >> k;
    st.push(k-1);
    for(short i=0; i<n; i++)
        for(short j=0; j<n; j++)
            in >> a[i][j];
    while(!st.empty()){
        short t = st.top();
        for(short i=n; i>=0; --i)
            if(a[t][i] &&
               find(v.begin(), v.end(), i)==v.end())
                {
                    st.push(i);
                }
        if(find(v.begin(), v.end(), t)==v.end()){
            v.push_back(t);
        } else st.pop();
    }
    for(short i=0; i<v.size(); i++)
        out << v[i]+1 << " ";
    return 0;
}

```

Observăm utilizarea metodelor *push*, *pop* și *top* specifice stivei și a algoritmului *find()* din header-ul *<algorithm>*.

Problema 6. Scrieți un program care afișează parcurgerea BFS (breadth first search) a unui grah dat prin matricea de adiacență. De exemplu:

graph.in	bfs.out	
8 1 0 1 1 1 0 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 0 0 1 0 0 0 0 0 1 0 0 0 1 1 0 0 0 1 0 0 0 0 1 0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 0 0 1 0 0	1 2 3 4 5 6 7 8	

Acest algoritm traversează grăful în lățime. Începând cu vârful inițial, se vor vizita toți vecinii nevizitați ai acestuia, după aceea toți vecinii nevizitați ai fiecărui dintre ei și.m.d. Algoritmul folosește o coadă (engl. queue):

```
ALGORITM_BFS(Graf G, Vârf y)
    Initializează coada Q cu {y}
    While (Q NOT Null) Do
        k ← Q.pop()           // returnarea și stergerea primului element din Q
        If (k not marked) Then
            MarkAndProcess k
            Q.push(all not marked Successors from k)
        End_If
    End_While
ALGORITM_BFS(Graf G, Vârf y)
```

Modelăm algoritmul de mai sus folosind containerul adaptor `std::queue`.

Program 6

```
#include <queue>
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;
int main(){
    std::ifstream in("graph.in");
    std::ofstream out("bfs.out");
    std::queue<short> qu;
    std::vector<short> v;
    short n, m, t;
    in >> n >> k;
    qu.push(k-1);
    std::vector<std::vector<bool> >
        a(n, vector<bool>(n, true));
    for (short i = 0; i < n; i++)
        for (short j = 0; j < n; j++)
            in >> t;
            a[i][j] = t ? true : false;
    }
    while (!qu.empty()){
        short t = qu.front();
        for (short i = 0; i < n; ++i)
            if (a[t][i] &&
                find(v.begin(), v.end(), i) == v.end())){

```

```

        qu.push(i);
    }
    if(find(v.begin(), v.end(), t)==v.end())){
        v.push_back(t);
    } else qu.pop();
}
for(size_t i=0; i<v.size(); i++)
    out << v[i]+1 << " ";
return 0;

```

Observăm că diferențele față de programul anterior nu sunt mari. Prima dintre ele este includerea `<queue>` în locul `<stack>`. Metodele `push`, `pop` și `empty` au aceeași semnificație ca și la `stack`. Metoda pentru extragerea unui element din coadă este `front` (la stivă este `top`). Ca variație și pentru a prezenta o nouă modalitate de a declara dinamic un tablou bidimensional, matricea `a` este considerată a fi „vector de vectori cu elemente de tip `bool`”, care, la declarare, este și inițializată cu $n \times n$ elemente `true`. Pentru a reda prima parcurgere *BFS* în ordine lexicografică, folosim în instrucțiunea `for` evidențiată căutarea vecinilor de la 0 la $n - 1$. Notă: rezolvări C ale problemelor 5 și 6 se găsesc de exemplu în [Log06].

Problema 7. Determinați mulțimea și multumulțimea literelor dintr-un citat dat, fără să se facă distincție între litere mari și mici. Exemplu:

citat.in	multime.out
Problema e ca dacă nu răsti, răsti chiar mai mult.	Multime: A B C D E H I L M N O P R S T U Multimultime: A A A A A A B C C C D E E H I I I I I I L L M M M M N O P R R R R
Cel ce nu stie încercă vrea, nu nu se miră că nu ajunge nicaieri.	Multime: A C E G I J L M N O R S T U V Multimultime: A A A A A C C C C E E E E E E E G I I I I I I J L M N N N N N N O O R R R R S S S S T T U U U U V

```

        aux /= i;
    }
}
}

for(mIt=m.begin(); mIt != m.end(); mIt++){
    out << mIt->first << " ";
    out << mIt->second << std::endl;
}

return 0;
}

```

Problema 9. Scrieți un program care pentru un număr natural dat n ($1 \leq n \leq 10$) generează toate permutările mulțimii $\{1, 2, \dots, n\}$ în ordine lexicografică. Exemplu:

n.in	perm.out																		
3	<table> <tr><td>1</td><td>2</td><td>3</td></tr> <tr><td>1</td><td>3</td><td>2</td></tr> <tr><td>2</td><td>1</td><td>3</td></tr> <tr><td>2</td><td>3</td><td>1</td></tr> <tr><td>3</td><td>1</td><td>2</td></tr> <tr><td>3</td><td>2</td><td>1</td></tr> </table>	1	2	3	1	3	2	2	1	3	2	3	1	3	1	2	3	2	1
1	2	3																	
1	3	2																	
2	1	3																	
2	3	1																	
3	1	2																	
3	2	1																	

Vectorul se completează mai întâi cu valorile $1, 2, \dots, n$. Folosim metoda `next_permutation()` din header-ul `<algorithm>`, care returnează `true`, dacă secvența specificată are o permutare succesoare, rază în care este și transformată corespunzător.

Program 9

```

#include <vector>
#include <iostream>
#include <algorithm>

int main(){
    short n, i;
    std::vector<short> v;
    std::ifstream in("n.in");
    std::ofstream out("perm.out");
    if(in && !in.eof() && in>>n){
        for(i=1; i<=n; i++) v.push_back(i);
        do{
            for(i=0; i<v.size(); i++){
                out.width(3);
                out << v[i];
            }
        }
    }
}

```

```

    out << std::endl;
    while(next_permutation(v.begin(), v.end()));

    return 0;
}

```

Ca exercițiu, modificați programul astfel încât permutările să fie afișate în ordine lexicografică, începând cu cea mai mare dintre ele și folosind metoda `next_permutation()`.

Problema 10. Ciurul lui Eratostene este o metodă foarte cunoscută pentru determinarea numerelor prime și a fost deja prezentată în primul capitol. Aşa cum specifică și numele, dintre numerele naturale se vor „cerne” acelea care nu sunt prime. Prin sită vor fi eliminati mai întâi multiplii lui 2, apoi multiplii lui 3, 5, 7 și.a.m.d. Numerele prime care nu sunt multipli ai altor numere vor rămâne la sfârșit în sită. Scrieți un program care verifică primalitatea pentru mai multe numere naturale din intervalul [1..5.000.000]. Exemplu:

numere.in	prim.out
123 345 12 101 43 284563 17 1006123 5000000 1021	123 -- compus 345 -- compus 12 -- compus 101 -- prim 43 -- prim 284563 -- compus 17 -- prim 1006123 -- prim 5000000 -- compus 1021 -- prim

Pentru manipularea cu biti este folosită clasa `bitset`. Aceasta este o clasă cu N biți și oferă metode pentru manipularea lor. Mai întâi, toți biții de la 0 la `MAX_SIZE` sunt setați pe 1 folosind metoda `set()` (metoda `reset()` setează toți biții pe 0). Apoi, se vor parcurge toate numerele naturale de la 2 la $\sqrt{MAX_SIZE}$ inclusiv și, dacă numărul corespunzător este prim (`sieve[i]` = 1), să vor „elimina” din sită toți multiplii lui prin setarea `sieve[i * j] = false`.

Program 10

```

#include <iostream>
#include <bitset>

const unsigned MAXSIZE = 5000000;

```

Vom folosi containerele asociative `std::set` (multime) și `std::multiset` (multimulțime). Toate literele citite se vor introduce în două containere și apoi le vom afișa în fișierul de ieșire folosind iteratori constanti (aceștia se recomandă pentru că elementele referențiate de ele nu își schimbă valorile). Programul este sugestiv în acest sens.

Program 7

```
#include <set>
#include <fstream>

int main(){
    std::ifstream in("citat.in");
    std::ofstream out("multimi.out");

    std::set<char> s;
    std::set<char>::const_iterator sIt;
    std::multiset<char> m;
    std::multiset<char>::const_iterator mIt;
    char c;

    while(in && !in.eof() && in>>c){
        if(isalpha(c)){
            s.insert(toupper(c));
            m.insert(toupper(c));
        }
    }

    out << "Multime: " << std::endl;
    for(sIt=s.begin(); sIt != s.end(); sIt++)
        out << *sIt << " ";
    out << "\n\nMultimulțime: " << std::endl;
    for(mIt=m.begin(); mIt != m.end(); mIt++)
        out << *mIt << " ";
}

return 0;
}
```

Problema 8. Scrieți un program care pentru un număr natural dat n ($1 \leq n \leq 10.000$) determină descompunerea în factori primi a lui $n!$, ca în exemplul:

numar.in	factorial.out
45	2^41 3^21 5^10 7^6 11^4

numar.in	factorial.out
	13^3
	17^2
	19^2
	23^1
	29^1
	31^1
	37^1
	41^1
	43^1

Pentru stocarea descompunerii în factori primi, vom folosi containerul asociativ `std::map`, care conține perechi (p, m) , în care p sunt numere prime, iar m sunt puterile acestora în descompunerea în factori primi a lui $n!$. Pentru fiecare factor prim i al tuturor numerelor de la 2 la n , dacă acesta nu se află deja în map (`m.find(i) == m.end()`), atunci este adăugată acesteia perechea $(i, 1)$; este prima apariție a acestui factor în produs. Altfel, se incrementează valoarea cu cheia i :

```
mit = m.find(i);
++(mit->second);
```

Program 8

```
#include <map>
#include <fstream>

int main(){
    typedef std::pair<int, int> TIntPair;

    std::ifstream in("numar.in");
    std::ofstream out("factorial.out");
    std::map<int, int> m;
    std::map<int, int>::iterator mit;
    int n, aux, i, k;

    if( in && !in.eof() && in>>n){
        for(int k=2; k<=n; k++){
            int aux = k;
            for(int i=2; aux>1 && i<=k; ++i){
                while(aux% i == 0){
                    if(m.find(i) == m.end())
                        m.insert(TIntPair(i, 1));
                    else {
                        mit = m.find(i);
                        ++(mit->second);
                    }
                }
            }
        }
        out << m;
    }
}
```

```

using namespace std;
int main(){
    unsigned long i, j, n;
    bitset<MAXSIZE+1> sieve;
    ifstream fin("numere.in");
    ofstream fout("prime.out");
    sieve.set(); sieve.at(1)=false;
    for(i=2; i*i<=MAXSIZE; i++)
        if(sieve[i]){
            j=2;
            while(i*j<=MAXSIZE){
                sieve[i*j]=false;
                j++;
            }
        }
    while(fin && !fin.eof() && fin>>n){
        if(sieve[n]) fout << n << " -- prim\n";
        else fout << n << " -- compus\n";
    }
    return 0;
}

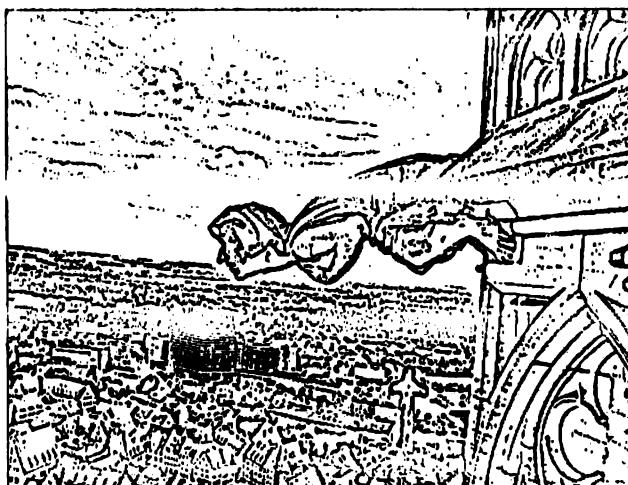
```

Notă: o rezolvare în C se găsește de exemplu în [Log06], p. 142.

Întrebări, exerciții, probleme propuse

1. Ce este *STL*? Care este scopul creării sale și ce conține?
2. Ce este programarea generică? Dați exemple.
3. Ce este un container în *STL*? Enumerați metodele generale asupra tuturor containerelor.
4. Enumerați containerele secvențiale, asociative, adaptor și specializate, precum și trăsăturile fiecăruiu dintre ele.
5. Ce sunt iteratorii și care este scopul lor? Cum sunt ei legați de container și algoritmi?
6. Ce este un functor? Definiți conceptele: *generator*, *predicat* și *predicat binar*. Dați exemple de utilizare a predicatului binar.
7. Header-ul `<algoritm>` din *STL* conține 6 clase de algoritmi. Cum își poate fi clasificat? Dați exemple de metode din fiecare categorie.
8. Modificați programul 4 astfel încât afișarea să se facă folosind iteratori (variantele *iterator* și *reverse_iterator*) și folosind metoda *at()* (acces verificat) în locul metodei *copy*.
9. Scrieți variante C pentru problemele 5 și 6 (parcurgerile *BFS* și *DFS*).

10. Dezvoltați programul 8 astfel încât să rezolve problema determinării combinațiilor de n luate câte k ($0 \leq k \leq n \leq 1000$), folosind descompuneri ca `std::map` și redând soluția în același format ca și factorialul, prin descompunerea în factori primi în ordinea crescătoare a acestora.
11. Modificați programul 9 astfel încât permutările să fie afișate în ordine colexicografică, începând cu cea mai mare dintre ele și folosind metoda `prev_permutation()` din `<algorithm>`.
12. Un `std::bitset<N>` se diferențiază de `std::vector<bool>` prin faptul că are o lungime fixă. Modificați programul 10 astfel încât să folosiți `std::vector<bool>` în loc de `std::bitset`.
13. Modificați toate programele care folosesc tipul `std::vector`, prin înlocuirea acestuia cu tipul `std::list`.
14. Manipularea sirurilor de caractere în C++ este mai ușoară decât în C, deoarece lucru cu memoria este rezolvat implicit, dar *STL* oferă și multe metode utile specifice. Cititi din *Help* metodele și utilizarea tipului sir de caractere `std::string`. Scrieți eventual un referat care să conțină teorie și exemple privind metodele uzuale.
15. Scrieți un program care stochează în variabile de tip `std::bitset` numere întregi date de la tastatură (de exemplu, 12, 567, -34, 5, -3421). Scrieți pentru aceasta două variante (una dintre ele folosind operatorii pe biți).
16. Scrieți un program demonstrativ pentru testarea comportării algoritmilor `for_each`, `search`, `transform`, `replace`, `reverse` și `swap`.



Vedere din turnul Catedralei din Ulm

Capitolul 4

Cutii ascunse

- **Descrierea problemei**
- **Analiza problemei și proiectarea soluției**
- **Algoritmul**
- **Programul**
- **Analiza programului**
- **Trei mici trucuri de programare**
- **Probleme propuse**
- **Observații**

Pericolul trecutului era acela că oamenii devineau sclavi.

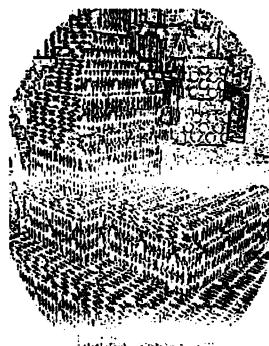
Pericolul viitorului este acela că ei pot deveni roboți.

Anonim

În acest capitol, prezentăm o problemă de programare dinamică. După descrierea problemei, urmărează analiza ei și proiectarea unei soluții, care este descrisă și în pseudocod. Apoi implementăm un program C++, care va fi analizat din perspectiva programării orientate obiect.

Descrierea problemei

Considerăm o cutie n -dimensională. Atunci când ne referim la două dimensiuni, perechea $(2, 3)$ poate să reprezinte, de exemplu, o cutie cu lățimea 2 și lungimea 3. Într-un spațiu tridimensional, tripletul $(4, 8, 9)$ poate să reprezinte o cutie cu lățimea 4, lungimea 8 și înălțimea 9. Nu este ușor să ne imaginăm o cutie în mai multe dimensiuni, dar vom putea abstractiza și opera cu acest concept. Spunem că o cutie $A = (a_1, a_2, \dots, a_n)$ încape în cutia $B = (b_1, b_2, \dots, b_n)$, dacă există o permutare π a mulțimii $\{1, 2, \dots, n\}$, astfel încât $a_{\pi(i)} < b_i$, pentru toți $i \in \{1, 2, \dots, n\}$. Aceasta înseamnă că ordinea dimensiunilor unei cutii poate fi schimbată aleatoriu (cutia se poate contorsiona). Dorim găsirea celui mai lung subșir de cutii, care încap una în alta. Cutile C_1, C_2, \dots, C_k reprezintă un astfel de subșir atunci când fiecare cutie C_i încape în cutia C_{i+1} (pentru orice i astfel încât $1 \leq i < k$). De exemplu, cutia $A = (2, 6)$ încape în cutia $B = (7, 3)$, deoarece dimensiunile cutiei A pot fi permuteate $A = (6, 2)$ și fiecare dimensiune este mai mică decât cea corespunzătoare în B . Cutia $A = (9, 5, 7, 3)$ nu încape în cutia $B = (2, 10, 6, 8)$, deoarece nu există nici o permutare a dimensiunilor cutiei A care să îndeplinească această condiție. Dar cutia $C = (9, 5, 7, 1)$ încape în cutia B , deoarece îi putem permuta dimensiunile la $(1, 9, 5, 7)$ și fiecare dintre ele este mai mică decât dimensiunea corespunzătoare din B .



Date de intrare. În fișierul *cutii.in* se găsesc mai multe șiruri de cutii. Fiecare astfel de șir începe cu o linie pe care se află numărul k de cutii și dimensiunea n a acestora. Următoarele k linii conțin fiecare căte n dimensiuni corespunzătoare unei cutii. Dimensiunea maximă a unei cutii este 250, cea minimă 1. Numărul maxim de cutii într-o secvență dată este 300. Considerăm că datele de intrare sunt corecte!

Date de ieșire. Trebuie deci să găsim un subșir cu număr maxim de cutii care încap una în alta. Pentru cazul când există mai multe subșiruri cu același număr maximal de cutii, se cere doar unul, în forma din *cutii.out*. Exemplu:

<i>cutii.in</i>	<i>cutii.out</i>
5 2	Lungime: 4
3 7	-----
8 10	3 1 4 5
5 2	*****
12 7	
21 18	Lungime: 4
8 6	-----
5 2 20 1 30 10	7 2 5 8
23 15 7 9 11 3	*****
40 50 34 24 14 4	
9 10 11 12 13 14	Lungime: 5
31 4 18 8 27 17	-----
44 32 13 19 41 19	5 4 2 7 9
1 2 3 4 5 6	*****
80 37 47 18 21 9	
9 5	
7 14 2 1 3	
49 80 15 50 10	
90 53 17 60 11	
4 3 2 15 10	
1 2 3 4 5	
6 7 8 9 10	
3 2 1 14 9	
92 54 65 19 15	

(ACM Internet Programming Contest 1990, Problem D. Stacking Boxes)

Analiza problemei și proiecțarea soluției

Propoziția 1. Se consideră două cutii n -dimensionale $A = (a_1, a_2, \dots, a_{n-1}, a_n)$ și $B = (b_1, b_2, \dots, b_{n-1}, b_n)$, cu proprietatea că $a_i \leq a_{i+1}$, $b_i \leq b_{i+1}$ pentru orice i de la 1 la $n-1$ (dimensiunile sunt sortate crescător). Atunci, cutia A încape în cutia B dacă și numai dacă $a_i < b_i$ pentru orice $i \in \{1, 2, \dots, n\}$.

Demonstrație. Folosim metoda reducerii la absurd. Presupunem că A încape în B și că există un $k \in \{1, 2, \dots, n\}$, astfel încât $a_k \geq b_k$. Considerăm cel mai mic număr natural k cu această proprietate: $a_i < b_i$ pentru toți $i \in \{1, 2, \dots, k-1\}$ și $a_k \geq b_k$. Deoarece (a_i) este un șir crescător, rezultă că $a_i \geq b_k$ pentru toți $i \in \{k+1, \dots, n\}$. Singura posibilitate ca la poziția k să alăbu loc $a_k < b_k$ este interschimbarea elementului a_k cu una dintre valorile $\{a_1, a_2, \dots, a_{k-1}\}$. Considerăm că aceasta este la poziția j . Pentru j va avea loc atunci $a_j \geq b_j$, de unde rezultă că A nu încape în B . Contradicției în cealaltă direcție, propoziția este adevărată prin definiție. \square

Primul pas în proiectarea algoritmului este deci sortarea crescătoare a dimensiunilor fiecărei cutii. Al doilea pas va fi ordonarea lexicografică a cutiilor, cu memorarea pozițiilor inițiale. Pentru ca o cutie A să încapă într-o altă cutie B , este necesar (nu și suficient), ca A să se afle înaintea lui B în acest șir ordonat lexicografic (o cutie de pe o poziție mai mică poate să încapă într-o cutie de pe o poziție mai mare, dar invers este imposibil). După aceste prelucrări, problema se reduce la determinarea subșirului maximal crescător. Condiția de comparație \leq devine acum încape . Pentru primul set de date de intrare din *cutii.in*, se vor executa următorii pași:

1. Ordinarea crescătoare a dimensiunilor pentru fiecare cutie	2. Sortarea lexicografică a cutiilor cu memorarea poziției inițiale																				
<table border="1"> <tr><td>3 7</td></tr> <tr><td>8 10</td></tr> <tr><td>5 2</td></tr> <tr><td>12 7</td></tr> <tr><td>21 18</td></tr> </table> <table border="1"> <tr><td>3 7</td></tr> <tr><td>8 10</td></tr> <tr><td>2 5</td></tr> <tr><td>7 12</td></tr> <tr><td>19 21</td></tr> </table>	3 7	8 10	5 2	12 7	21 18	3 7	8 10	2 5	7 12	19 21	<table border="1"> <tr><td>3 7</td></tr> <tr><td>8 10</td></tr> <tr><td>2 5</td></tr> <tr><td>7 12</td></tr> <tr><td>18 21</td></tr> </table> <table border="1"> <tr><td>2 5 (3)</td></tr> <tr><td>3 7 (1)</td></tr> <tr><td>7 12 (4)</td></tr> <tr><td>8 10 (2)</td></tr> <tr><td>18 21 (5)</td></tr> </table>	3 7	8 10	2 5	7 12	18 21	2 5 (3)	3 7 (1)	7 12 (4)	8 10 (2)	18 21 (5)
3 7																					
8 10																					
5 2																					
12 7																					
21 18																					
3 7																					
8 10																					
2 5																					
7 12																					
19 21																					
3 7																					
8 10																					
2 5																					
7 12																					
18 21																					
2 5 (3)																					
3 7 (1)																					
7 12 (4)																					
8 10 (2)																					
18 21 (5)																					

Determinarea subșirului maximal crescător este o problemă clasică de programare dinamică și este tratată și în capitolul 11. Pentru exemplul nostru, subșirul crescător maxim și cu condiția că încape cutie $(2, 5) \rightarrow (3, 7) \rightarrow (7, 12) \rightarrow (18, 21)$, lungimea că lungimea 4. Pozițiile inițiale ale cutiilor corespunzătoare $(3 1 4 5)$ se vor scrie în fișierul de ieșire.

Algoritmul

Considerăm n obiecte de tip *Cutie* C_1, \dots, C_n , care își cunosc dimensiunile și poziția inițială. Pentru fiecare i considerăm un subșir de cutii că are ca ultim element cutia C_i . Pentru acesta, vom construi doi vectori $v[]$ și $vPred[]$. Lungimea subșirului care sfârșește cu C_i este salvată în $v[i]$ și $vPred[i]$ este indexul penultimei cutii în acest subșir ($vPred[i] = -1$ și $v[i] = 1$). Deci:

- $v[i]$ este lungimea maximă a unui subșir de cutii care încap una în alta, a cărui

ultimă cutie este C_i . Formal, scriem:

- $v[1] \leftarrow 1$ (prima cutie nu poate să conțină alta, acest subșir este format numai din acest element),
- $v[i] \leftarrow 1 + \max\{v[j] \mid j < i \text{ și } C_j \text{ începe în } C_i\}$ (C_i este penultima cutie în acest subșir).

- $vPred[i]$ conține indexul j cu condiția: C_j să fie cutia predecesoare în sirul maximă în care ultima cutie este C_i (dacă o cutie nu are nici o predecesoare, atunci aceasta valoare este -1):
 - $vPred[1] \leftarrow -1$ (prima cutie nu are nici o predecesoare),
 - $vPred[i] \leftarrow j$, $v[j]$ maximal cu ($j < i$ și C_j începe în C_i).

Dacă există mai multe subșiruri cu aceeași lungime maximală, atunci se va considera doar primul dintre ele.

Vectorii $v[]$ și $vPred[]$ vor fi populati sevențial. Folosim variabila $imax$ pentru a salva indexul actual optimal. Dacă lungimea $v[i]$ corespunzătoare poziției actuale este mai mare decât valoarea $v[imax]$, atunci $imax$ va fi actualizat.

În acest moment putem formula algoritmul în pseudocod:

```

ALGORITM_CUTII_ASCUNSE
 1. Read Boxes  $C_1, C_2, \dots, C_n$ 
 2. For ( $i \leftarrow 1, n$ ; step 1) Execute
    Sort_Dimensions( $C_i$ )
   End_For
 3. Sort Lexicographical( $C_1, C_2, \dots, C_n$ )
 4.  $v[1] \leftarrow 1$ ,  $vPred[1] \leftarrow -1$ ,  $imax \leftarrow 1$ 
 5. For ( $i \leftarrow 2, n$ ; step 1) Execute
    5.1.  $v[i] \leftarrow 1$ ,  $vPred[i] \leftarrow -1$ 
    5.2. For ( $j \leftarrow 1, i-1$ ; step 1) Execute
       $v[i] \leftarrow v[j]+1$ 
       $vPred[i] \leftarrow j$ 
    End_If
   End_For
 5.3. If ( $v[i] > v[imax]$ ) Then
     $imax \leftarrow i$ 
  End_If
 End_For
 6. recoverBoxesSubstring ( $C[1..v[1]], vPred[1..imax]$ )
END_ALGORITM_CUTII_ASCUNSE
```

Metoda *recoverBoxesSubstring()* reconstituie recursiv subșirul maximal de cutii pe baza vectorului *vPred[]* și a indexului *imax*:

```

recoverBoxesSubstring (C[1..n], vPred[], i)
  1. If (vPred[i] ≠ -1) Then
    recoverBoxesSubstring(C[1..n], vPred[], vPred[i])
    End_If
  2. Write Original_Position(C)
End_recoverBoxesSubstring ()

```

Complexitatea algoritmului este $O(n^2m + nm \log m)$, unde n reprezintă numărul cutiilor și m dimensiunea unei cutii (din pasul 2 rezultă $O(nm \log m)$), deoarece sunt sortate n cutii; din pasul 5 rezultă $O(n^2)$, deoarece avem două bucle *for* imbricate, fiecare de lungime n ; în a doua buclă *for* avem nevoie de $O(m)$ pentru a testa condiția *in cape*.

Programul

Pentru a implementa modelul matematic descris, scriem clasa *CBox*, care conține ca membri *_n* (dimensiunea cutiei), *_i* (poziția inițială) și vectorul *_vDim* (ce reține dimensiunile). Metodele *getI()*, *getN()*, *elementAt()*, *sort()* și *isIt()* preucrează elemente de tipul abstract *CBox*.

```

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

class CBox{
    short _n, _i;
    vector<short> _vDim;
public:
    CBox(vector<short> vDim, short n, short i):
        _vDim(vDim), _n(n), _i(i){}
    short getI() const {return _i;}
    short getN() const {return _n;}
    short elementAt(short i) const{
        return _vDim.at(i);}
};

```

```

void sort();
bool isFit(CBox *another) const;
friend bool operator>(CBox &b1, CBox &b2);
};

void CBox::sort(){
    std::sort( _vDim.begin(), _vDim.end() );
}

bool CBox::isFit(CBox *another) const{
    bool b = true;
    if(another->getN()!=_n) return false;
    for(short i=0; b && i<_n; i++)
        if(_vDim.at(i)>=another->elementAt(i))
            b = false;
    return b;
}

bool operator>(CBox &b1, CBox &b2){
    return
        lexicographical_compare(b2._vDim.begin(), b2._vDim.end(),
                               b1._vDim.begin(), b1._vDim.end());
}

istream& operator>>(istream &is, vector<CBox*> &v) {
    short n, k, j, i, aux;
    vcl::vector<CBox*> vDim;
    if( !is ) return is;
    is >> k >> n;
    v.clear();
    for(i=0; i<k; i++){
        vDim.clear();
        is >> aux; vDim.push_back(aux);
    }
    CBox *box = new CBox(vDim, n, i+1);
    box->sort();
    v.push_back(box);
}
return is;
}

bool isBoxSmaller(CBox* b1, CBox *b2){
    return !(b1 > b2);
}

```

Testează dacă începe în altă cutie (presupunem că dimensiunile cutiilor sunt sorteate crescător și folosim Propoziția 1):

- mărimi diferite → nu începe;
- când o dimensiune a cutiei este mai mare sau egală cu dimensiunea corespunzătoare în cealaltă cutie, atunci nu începe;
- altfel, începe.

istream& operator>>(istream &is, vector<CBox*> &v) {

```

short n, k, j, i, aux;
vcl::vector<CBox*> vDim;
if( !is ) return is;
is >> k >> n;
v.clear();
for(i=0; i<k; i++){
    vDim.clear();
    is >> aux; vDim.push_back(aux);
}
CBox *box = new CBox(vDim, n, i+1);
box->sort();
v.push_back(box);
}
return is;
}
```

Citire obiecte de cutii (supraîncărcarea operatorului „>>”):

- citește k (numărul de cutii) și n (dimensiunea unei cutii);
- citește cele n dimensiuni ale fiecărei cutii și le stochează în vectorul $vDim$;
- creează obiectul CBox cu ajutorul constructorului CBox ($vDim$, n , $i+1$), sortează dimensiunile cutiei cu ajutorul sort() și adaugă acest obiect vectorului v ; care va fi returnat.

```

void doProcess(vector<CBox*> &vCBoxes, vector<int>&v,
               vector<int>&vPred, int &imax){
    std::sort(vCBoxes.begin(), vCBoxes.end(), isBoxSmaller);
    v.clear(); vPred.clear();
    v.push_back(1);
    vPred.push_back(-1);
    imax = 0;
    int n=(int)vCBoxes.size();
    CBox *b1, *b2;
    for(int i=1; i<n; i++){
        v.push_back(1);
        vPred.push_back(-1);
        b1=(CBox*)vCBoxes[i];
        for(int j=0; j<i; j++){
            b2=(CBox*)vCBoxes[j];
            if(b2->isFit(b1) &&
               v[j]+1 > v[i]){
                v[i] = v[j] + 1;
                vPred[i] = j;
            }
        }
        if(v[i] > v[imax]) imax = i;
    }
}

void recoverBoxesSubstring(vector<CBox*> &vCBoxes,
                           vector<int>&vPred,
                           int i, string& out){
    if(vPred[i]+1)
        recoverBoxesSubstring(vCBoxes, vPred, vPred[i], out);
    CBox* b = (CBox*)vCBoxes[i];
    out << b->getI() << " ";
}

int main(){
    vector<CBox*> vCBoxes;
    vector<int> v, vPred;
    int imax;
    ifstream in("cutii.in");
    ofstream out("cutii.out");
    while(in.&& !in.eof() && in >> vCBoxes){
        doProcess(vCBoxes, v, vPred, imax);
        out << "Lungime: " << v[imax] << endl;
        out << "-----" << endl;
        recoverBoxesSubstring (vCBoxes, vPred, imax, out);
        out << endl;
    }
}

```

1. Sortarea lexicografică a cutiilor.
2. Pentru fiecare cutie, creăm $v[]$ și $vPred[]$ cu semnificațiile:
 - $v[0] \leftarrow 1$;
 - $v[i] \leftarrow 1 + \max\{v[j] \mid j < i \text{ și cutia } j \text{ începe în cutia } i\}$ ($v[i]$ este lungimea maximală a unui subșir cu i ultima cutie);
 - $vPred[0] \leftarrow -1$ (prima cutie nu are nici un predecesor).
3. Dacă există o cutie j cu proprietatea că ea începe în cutia i , atunci: $vPred[i]$ este indexul j al cutiei cu $v[j]$ maximal; dacă o astfel de cutie nu există, atunci $vPred[i] \leftarrow -1$.

```

    out << "*****" << endl;
}
return 0;
}

```

Analiza programului

1. Tipul abstract *CBox* este reprezentarea în C++ a conceptului de *cutie* prezentat în descrierea problemei. Clasa încapsulează date și metode ce prelucrează într-un mod natural aceste date.
2. Metoda *doProcess()* sortează lexicografic întregul vector cu cutii. Pentru a realiza aceasta, folosim metoda *sort()* din *Standard Template Library (STL)*, antetul *<algorithm>*. Pe lângă pointerii care mărginesc secvența de sortat (*vCBoxes.begin()* și *vCBoxes.end()*), remarcăm folosirea unui al treilea parametru, obiectul *isBoxSmaller()*, de tip funcție. Acesta definește criteriul de comparație, care trebuie să fie satisfăcut de două elemente consecutive din secvență sortată. Un astfel de predicat binar este definit de către utilizator, are două argumente și returnează *true* atunci când criteriul este adevărat și *false*, în caz contrar.

```
std::sort(vCBoxes.begin(), vCBoxes.end(), isBoxSmaller);
```

3. Metoda *std::sort()* va fi folosită din nou, de această dată cu doi parametri:

```

void CBox::sort()
{
    d::sort( vDim.begin(), _vDim.end());
}

```

Se presupune că criteriul de comparație este operatorul implicit „≤”.

4. În metoda *isBoxSmaller()*, observăm folosirea „>” pentru a compara două obiecte de tip *CBox*. Această operare nu poate fi realizată direct, prin urmare trebuie să se declară ca metodă prietenă – *friend* – locală a *operator>()*). Această metodă trebuie să fie declarată ca metodă prietenă – *friend* – a clasei *CBox*, deoarece ea are nevoie să acceseze membrul privat *_vDim*.

În metoda *operator>()*, observăm folosirea metodei *std::lexicographical_compare()*, care se află tot în *header-ul <algorithm>* al *STL*. Această metodă compară fiecare element al primei secvențe cu elementul corespunzător al celei de-a doua secvențe, pentru a decide care este mai mică lexicografic.

Trei mici trucuri de programare

1. Apelul repetat al aceleiași funcții în condiția unei bucle

Exemplu. Dacă dorim prelucrarea elementelor unui vector:

```
vector<int>::iterator itBegin = v.begin();
vector<int>::iterator it;
for(it = v.end()-1; it != itBegin; --it)...
```

În acest caz, este mai bine să folosim, ca mai sus, o nouă variabilă *itBegin* și să o inițializăm cu *V.begin()*, decât să folosim mereu apelul funcției *V.begin()*:

```
vector<int>::iterator it;
for(it = v.end()-1; it != v.begin(); --it)
```

Motivul este că, în prima variantă, apelul funcției *begin()* este executat doar o dată, iar pentru containere de dimensiune mai mare salvează timp.

2. Condiție într-o expresie booleană

De multe ori, folosim în carte expresii de forma **if(0==n)..., while(i==i && 5==j)** etc., în locul expresiilor „naturale” **if(n==0)..., while(i==1 && j==5)** etc. Ultimele sunt echivalente cu primele, dar este recomandată utilizarea primei forme. Deseori se uită un semn de egalitate și, în cazul primei forme, compilatorul va indica acest lucru pe când, în al doilea caz, va compila, dacă erorii nu vor fi același (atribuire în loc de compararea egalității).

3. Instrucțiune break într-o instrucțiune de selecție multiplă switch

Ambele utilizări de mai jos sunt echivalente:

```
switch((p2-p5)%4){
    case 0: uc = (uc*6)%10; break;
    case 1: uc = (uc*2)%10; break;
    case 2: uc = (uc*4)%10; break;
    case 3: uc = (uc*8)%10; break;
}
```

```
switch((p2-p5)%4){
    case 0: uc = (uc*6)%10; break;
    case 1: uc = (uc*2)%10; break;
    case 2: uc = (uc*4)%10; break;
    case 3: uc = (uc*8)%10;
}
```

De obicei, este mai bine să se încheie ultimul caz de decizie tot cu o instrucțiune *break*. Dacă ulterior se dezvoltă programul și apare un nou caz în instrucțiunea *switch*, atunci nu este posibilă neintroducerea lui din neatenție.

Probleme propuse

1. Într-un sir ordonat lexicografic de cutii (ale căror dimensiuni sunt sortate), o condiție necesară pentru ca o cutie *A* să încapă într-o cutie *B* este ca *A* să se afle înaintea lui *B* în sir. De ce nu este această condiție și suficientă? Găsiți un contraexemplu.
2. Eliminați declarația de funcție prietenă (*friend*) a metodei *operator>*() din clasa *CBox*. Ce eroare furnizează compilatorul? Modificați programul astfel încât această metodă să fie declarată doar extern local.
3. Implementați iterativ metoda *recoverBoxesSubstring()*.
4. Se poate întâmpla ca datele de intrare să nu fie corecte (caracter false, lungimea unui sir de cutii prea mare sau negativă etc.). Dezvoltați programul astfel încât să se verifice și corectitudinea datelor de intrare.
5. Scrieți un program care generează fișiere de intrare cu mai multe cazuri și folosiți-le în program. Scrieți o metodă intelligentă, care pentru dimensiuni mai mari să aibă și soluții subșiruri de dimensiuni mai mari.
6. Se poate ca pentru o soluție de lungime maximală să existe mai multe subșiruri de cutii care au aceeași lungime. Găsiți toate subșirurile de lungime maximală.
7. Scrieți un program C care rezolvă problema, pentru a compara programarea procedurală cu cea orientată obiect.
8. Găsiți motivele pentru care algoritmul *Cutii_Ascunse* este unul de programare dinamică, și nu unul de tip *Creativ*, *Backtracking* sau *Divide-et-Impera*.

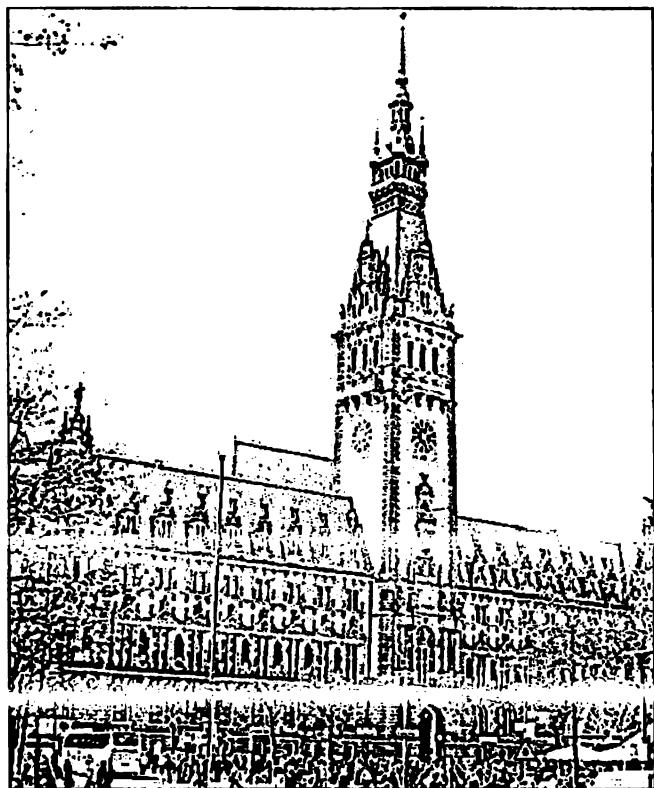
Observații:

În acest capitol am folosit:

1. metoda programării dinamice;
2. recursivitatea;

În cadrul acestui capitol am folosit și clasele și interfațele operatorilor, funcții prietene (*friend*), funcții membru *const*, polimorfie;

4. clasa *container std::vector* din *STL* din antetul *<vector>*, cu metodele: *pop_back()*, *push_back()*, operatorul de acces *[], clear(), begin(), end(), at()*;
5. metode din *STL*, antetul *<algorithm>. sort(), lexicographical compare();*
6. fluxuri de intrare și de ieșire pentru citirea și scrierea în/din fișiere.



Rathaus din Hamburg (construită între 1886 și 1897)

Capitolul 5

Sume de puteri

- **Descrierea problemei**
- **Analiza problemei. Modelarea algebrică**
- **De la formula recursivă spre algoritm**
- **Algoritmul**
- **Programul**
- **Probleme propuse**

Abia atunci când ne îndeplătim de munți
și putem vedea în adevărta lor măretie; așa e și cu prietenii...
Hans Christian Andersen

Descrierea problemei

Fie numărul natural $k \geq 0$. Suma de puteri de grad k pentru n se definește astfel:

$$S_k(n) = 1^k + 2^k + 3^k + \dots + n^k \quad (1)$$

Se poate demonstra (cu ajutorul inducției matematice și al formulei (13) de mai jos), că $S_k(n)$ este un polinom de grad $(k+1)$ în n , cu n coeficienți raționali, deci $S_k(n)$ se poate scrie sub forma:

$$S_k(n) = \frac{1}{M} (a_{k+1}n^{k+1} + a_k n^k + \dots + a_1 n + a_0) \quad (2)$$

M și $a_{k+1}, a_k, \dots, a_1, a_0$ sunt numere întregi, M este număr natural. Cu aceste condiții și cu condiția ca M să fie minimal, pentru fiecare număr natural k există exact o soluție $(M, a_{k+1}, a_k, \dots, a_1, a_0)$ a problemei. Se cere determinarea acestor soluții pentru un număr natural k dat.

Date de intrare. În fișierul `psum.in` se găsesc unul sau mai multe numere naturale k ($0 \leq k \leq 20$).

Date de ieșire. Scrieți în fișierul `psum.out` numerele $M, a_{k+1}, a_k, \dots, a_1, a_0$ pe căte o linie pentru fiecare caz de intrare, cu $/M$ între ele, ca în exemplul:

<code>psum.in</code>	<code>psum.out</code>
2	6 2 3 1 0
5	12 2 6 5 0 -1 0 0
0	1 1 0
16	510 30 255 680 0 -2380 0 8840 0 -24310 0 44200 0 -46988 0 23800 0 -3617 0

(ACM North-Eastern European Regional Programming Contest, 1997-1998)

Analiza problemelor. Modelarea algebrică

Primele sume $S_k(n)$ sunt:

$$S_0(n) = \sum_{i=1}^n i^0 = n \quad (3)$$

$$S_1(n) = \sum_{i=1}^n i = \frac{n(i+1)}{2} = \frac{1}{2}(n^2 + n) \quad (4)$$

$$S_2(n) = \sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6} = \frac{1}{6}(2n^3 + 3n^2 + n) \quad (5)$$

$$S_3(n) = \sum_{i=1}^n i^3 = \frac{n^2(n+1)^2}{4} = \frac{1}{4}(n^4 + 2n^3 + n^2) \quad (6)$$

$$S_4(n) = \sum_{i=1}^n i^4 = \frac{n(n+1)(2n+1)(3n^2 + 3n - 1)}{30} = \frac{1}{30}(6n^5 + 15n^4 + 10n^3 - n) \quad (7)$$

Aceste formule sunt relativ cunoscute. Ele pot fi demonstrate, de exemplu, prin inducție și ne dorim mai departe determinarea lui $S_5(n)$. Cu ajutorul binomului lui Newton obținem:

$$(i-1)^6 = i^6 - C_6^1 i^5 + C_6^2 i^4 - C_6^3 i^3 + C_6^4 i^2 - C_6^5 i + 1, \quad (8)$$

echivalent cu:

$$i^6 - (i-1)^6 = 6i^5 - 15i^4 + 20i^3 - 15i^2 + 6i - 1 \quad (9)$$

$$\sum_{i=1}^n (i^6 - (i-1)^6) = 6 \sum_{i=1}^n i^5 - 15 \sum_{i=1}^n i^4 + 20 \sum_{i=1}^n i^3 - 15 \sum_{i=1}^n i^2 + 6 \sum_{i=1}^n i - \sum_{i=1}^n 1 \quad (10)$$

Reducem termenii în membrul stâng:

$$n^6 = 6S_5(n) - 15S_4(n) + 20S_3(n) - 15S_2(n) + 6S_1(n) - S_0(n) \quad (11)$$

Din formula (11) se obține imediat:

$$S_3(n) = \frac{n^2(n+1)^2(2n^3 + 2n - 1)}{12} = \frac{1}{12}(2n^6 + 6n^5 + 5n^4 - n^2) \quad (12)$$

Generalizarea acestui exemplu concret ne conduce la:

$$(k+1)S_k(n) = n^{k+1} + C_{k+1}^1 S_{k-1}(n) - C_{k+1}^3 S_{k-2}(n) + \dots + (-1)^p C_{k+1}^p S_{k-p+1}(n) + \dots \\ + (-1)^k C_{k+1}^k S_1(n) + (-1)^{k+1} S_0(n) \quad (13)$$

De la formula recursivă spre algoritm

Pe baza formulei (13), se deduce că, pentru a calcula soluția $(M, a_{k+1}, a_k, \dots, a_1, a_0)$ a problemei pentru un k dat, este nevoie de soluțiile futurilor problemelor pentru dimensiuni mai mici decât k , anume $0, 1, \dots, k-1$. Toate aceste informații vor fi secvențial calculate și salvate, pentru a putea fi folosite ulterior. Vom folosi un tablou $M[]$ pentru a salva valorile $M[1], M[2], \dots, M[k]$ cu condiția: $M[i]$ este numărul natural M din descrierea problemei pentru $S_i(n)$, $0 \leq i \leq k$.

În tabloul bidimensional $A[][]$, salvăm valorile $(a_{k+1}, a_k, \dots, a_1, a_0)$, pentru $0 \leq i \leq k$, cu semnificația că linia i conține coeficienții corespunzători pentru $S_i(n)$.

Vizualizarea tabelară a matricei $A[][]$ și a vectorului $M[]$

$A[0][0]$	$A[0][1]$	$A[1][0]$	$A[1][1]$	$A[1][2]$							$M[0]$
$A[1][1][0]$	$A[1][1][1]$	$A[1][1][2]$									$M[1]$
\vdots	\vdots	\vdots	\vdots	\vdots							\vdots
$A[k-1][0]$	$A[k-1][1]$	$A[k-1][2]$	$A[k-1][3]$	$A[k-1][4]$	\dots	$A[k-1][k]$					$M(k-1)$
$A[k][0]$	$A[k][1]$	$A[k][2]$	$A[k][3]$	$A[k][4]$	\dots	$A[k][k]$	$A[k][k+1]$				$M(k)$

Care aspect reprezintă propoziția (2) rezolvată?

$$S_k(n) = \frac{1}{M[k]} (A[k][k+1]n^{k+1} + A[k][k]n^k + \dots + A[k][1]n + A[k][0]) \quad (14)$$

Înălțăm acum relația recursivă (13) pentru a determina secvența $(M[k], A[k][k+1], A[k][k], \dots, A[k][1], A[k][0])$ cu ajutorul secvențelor $(M[i], A[i][i+1], A[i][i], \dots, A[i][1], A[i][0])$ pentru $0 \leq i < k$. La prima vedere pare să fie foarte dificil, dar vom vedea cum prin transformări succesive formula se simplifică. Prin substituția cu tablouri a valorilor a_1, a_2, \dots, a_k și M din formula (13) obținem:

$$\begin{aligned}
 (k+1)S_k(n) &= n^{k+1} + \\
 &+ C_{k+1}^2 \left(\frac{1}{M[k-1]} (A[k-1][k]n^k + A[k-1][k-1]n^{k-1} + \dots + A[k-1][1]n + A[k-1][0]) \right) \\
 &- C_{k+1}^3 \left(\frac{1}{M[k-2]} (A[k-2][k-1]n^{k-1} + A[k-2][k-2]n^{k-2} + \dots + A[k-2][1]n + A[k-2][0]) \right) \\
 &+ \dots + \\
 (-1)^p C_{k+1}^p &\left(\frac{1}{M[k+1-p]} (A[k+1-p][k+2-p]n^{k+2-p} + \dots + A[k+1-p][1]n + A[k+1-p][0]) \right) \\
 &+ \dots \\
 (-1)^k C_{k+1}^k &\left(\frac{1}{M[1]} (A[1][2]n^2 + A[1][1]n + A[1][0]) \right) + \\
 (-1)^{k+1} C_{k+1}^{k+1} &\left(\frac{1}{M[0]} (A[0][1]n + A[0][0]) \right)
 \end{aligned} \tag{15}$$

Grupăm termenii după puterile lui n în forma unui polinom de grad $(k+1)$:

$$\begin{aligned}
 (k+1)S_k(n) &= n^{k+1} + \\
 n^k &\left(\frac{C_{k+1}^2}{M[k-1]} A[k-1][k] \right) + \\
 n^{k-1} &\left(-\frac{C_{k+1}^2}{M[k-1]} A[k-1][k-1] - \frac{C_{k+1}^3}{M[k-2]} A[k-2][k-1] \right) + \\
 n^{k-2} &\left(\frac{C_{k+1}^2}{M[k-1]} A[k-1][k-2] - \frac{C_{k+1}^3}{M[k-2]} A[k-2][k-2] - \frac{C_{k+1}^4}{M[k-3]} A[k-3][k-2] \right) + \\
 &\dots
 \end{aligned} \tag{16}$$

$$\begin{aligned}
 n^p &\left(\frac{C_{k+1}^2}{M[k-1]} A[k-1][p] - \frac{C_{k+1}^3}{M[k-2]} A[k-2][p] + \dots + (-1)^{k+2-p} \frac{C_{k+1}^{k+2-p}}{M[p-1]} A[p-1][p] \right) \dots \\
 n &\left(\frac{C_{k+1}^2}{M[k-1]} A[k-1][1] - \frac{C_{k+1}^3}{M[k-2]} A[k-2][1] + \dots + (-1)^{k+1} \frac{C_{k+1}^{k+1}}{M[0]} A[0][1] \right) + \\
 &\left(\frac{C_{k+1}^2}{M[k-1]} A[k-1][0] - \frac{C_{k+1}^3}{M[k-2]} A[k-2][0] + \dots + (-1)^{k+1} \frac{C_{k+1}^{k+1}}{M[0]} A[0][0] \right)
 \end{aligned}$$

În această formulă (16), coeficienții se pot scrie ca:

$$W[i] = (-1)^{k+1-i} \frac{C_{k+1}^{k+1-i}}{M[i]}, \text{ pentru orice } i, \text{ astfel încât } 0 \leq i \leq k-1 \tag{17}$$

Cu ajutorul formulei (17) formula (16) se transformă în:

$$\begin{aligned}
 (k+1)S_k(n) = & n^{k+1} + \\
 & n^k (W[k-1]A[k-1][k]) + \\
 & n^{k-1} (W[k-1]A[k-1][k-1] + W[k-2]A[k-2][k-1]) + \\
 & n^{k-2} (W[k-1]A[k-1][k-2] + W[k-2]A[k-2][k-2] + W[k-3]A[k-3][k-2]) + \\
 & \dots \\
 & n^p (W[k-1]A[k-1][p] + W[k-2]A[k-2][p]) + \dots + W[p-1]A[p-1][p]) + \\
 & \dots \\
 & n(W[k-1]A[k-1][1] + W[k-2]A[k-2][1]) + \dots + W[0]A[0][1]) + \\
 & (W[k-1]A[k-1][0] + W[k-2]A[k-2][0]) + \dots + W[0]A[0][0])
 \end{aligned} \tag{18}$$

În acest moment, am ajuns deja la o formă mai simplă. Coeficienții puterilor lui n în această egalitate pot fi salvați într-un tablou unidimensional cu fracții raționale $F[]$:

$$\begin{aligned}
 F[0] &= \sum_{t=k-1}^1 W[t]A[t][0] + W[0]A[0][0] \\
 F[p] &= \sum_{t=k-1}^{p-1} W[t]A[t][p] \text{ pentru orice } p \text{ cu } 1 \leq p \leq k \\
 F[k+1] &= 1
 \end{aligned} \tag{19}$$

Prin înlocuirea formulelor (19) în (18), se obține:

$$S_k(n) = \frac{1}{(k+1)Q} (F[k+1]n^{k+1} + F[k]n^k + F[k-1]n^{k-1} + \dots + F[1]n + F[0]) \tag{20}$$

Am ajuns în acest fel la o formulă care este foarte asemănătoare cu cea din descrierea problemei. Trebuie doar să mai ținem cont de determinarea soluției problemei cu F minimă. Aceasta și va obține prin calcularea celui mai mic multiplu comun al tuturor numitorilor fracțiilor $F[0], F[1], \dots, F[k], F[k+1]$. Fie acesta Q . Cu ajutorul său, vom obține din formula (20):

$$S_k(n) = \frac{1}{(k+1)Q} (F'[k+1]n^{k+1} + F'[k]n^k + F'[k-1]n^{k-1} + \dots + F'[1]n + F'[0]) \tag{21}$$

$$F'[r] = F[r] \cdot Q, 0 \leq r \leq k+1$$

Deoarece Q este cel mai mic multiplu comun al numitorilor $F[i]$, rezultă că $F'[i]$ conține doar numere întregi și $(k+1)Q$ este cel mai mic M pentru suma $S_k(n)$.

Algoritmul

Din reprezentarea algebraică de mai sus, rezultă că $M[0] = 1$, $A[0][1] = 1$, $A[0][0] = 0$.

Algoritmul este atunci:

```

ALGORITM_SUME_PUTERI
1. Read k (0 ≤ k ≤ 20)
2. genCombinations(C[k + 1][k + 1])
3. M[0] = 1, A[0][1] = 1, A[0][0] = 0
4. For(p ← 1, k; step 1) Execute
   4.1. sign = 1
   4.2. For(t ← p - 1, 0; step -1) Execute
      W[t] = simplifyFraction(sign * C[p + 1][p + 1 - t]/M[t])
      sign = sign * (-1)
   End_For
   4.3. For(t ← p, 0; step -1) Execute
      fAux = Fraction(0/1)
      For(r ← p - 1, t - 1; r ≥ 0; step -1) Execute
         fAux2 = simplifyFraction(W[r] * A[r][t])
         fAux = sumFractions(fAux, fAux2)
      End_For
      F[t] = fAux
   End_For
   4.4. fAux = simplifyFraction(W[0] * A[0][0])
   F[0] = sumFractions(F[0], fAux);
4.5. F[p+1] = Fraction (1/1)
4.6. M[p+1] = F[p+1]
4.7. M[p] = (p + 1) * Q
4.8. For(t ← p + 1, 0; step -1) Execute
   fAux2 = simplifyFraction (F[t] * Q)
   A[p][t] = Numerator(fAux2)
End_For
End_For
5. Write M[k], A[k][k+1], A[k][k], ..., A[k][1], A[k][0]
END_ALGORITM_SUME_PUTERI
```

- La pasul 2 se generează toți coeficienții binomiali până la nivelul $k + 1$ (conform egalității (13) pentru calculul lui $S_k(n)$, avem nevoie de coeficienții binomiali de

$k+1$). Acest calcul se bazează tot pe un algoritm de programare dinamică cu ajutorul formulelor:

$$C[i][0] = C[i][i] = 1 \text{ pentru } 0 \leq i \leq k + 1.$$

$$C[i][j] = C[i - 1][j - 1] + C[i - 1][j] \text{ pentru } 1 \leq i \leq k + 1, 1 \leq j < i$$

- La pasul 3 se inițializează valorile corespunzătoare lui $S_0(n)$ ((2) și (14)).
- La pasul 4 începe o buclă *for* în care sunt calculate secvențial valorile $A[p][0]$, $A[p][1]$, ..., $A[p][p+1]$, $M[p]$, $1 \leq p \leq k$, care sunt soluțiile problemei pentru $1, 2, \dots, k$, conform formulelor de mai sus.
- 4.1 și 4.2: fracțiile $W[]$ sunt calculate cu ajutorul formulelor (17). Metoda *simplifyFraction()* returnează fracția simplificată. Aceasta se realizează prin aflarea celui mai mare divizor comun d al numitorului și numărătorului, urmată de împărțirea acestora la d .
- Pasul 4.3 calculează fracțiile $F[]$ cu ajutorul formulei (19) și al metodei *sumFractions()* (ce returnează suma a două fracții raționale).
- Pasul 4.4. adună la $F[0]$ valoarea $W[0] \times A[0][0]$, conform primei părți a formulei (19).
- La fel, pe baza formulei (19), se inițializează fracția $F[p+1]$ cu $\frac{1}{1}$ la pasul 4.5.
- La pasul 4.6, se va calcula Q ca fiind cel mai mic multiplu comun al numitorilor fracțiilor $F[0], F[1], \dots, F[p], F[p+1]$ cu ajutorul metodei *lcmDenomin()*.
- La pasul 4.7, se calculează $M[p]$ conform (20).
- Iată, se vor calcula valorile $A[p][p+1], A[p][p], \dots, A[p][0]$ la pasul 4.8, cu ajutorul (20).

Complexitatea și corectitudinea algoritmului. Complexitatea algoritmului este $O(n^3)$ (pasul 4 conține trei bucle *for* imbricate), deci algoritmul este polinomial. El este corect, pentru că se bazăază pe succesiuni corecte de transformări matematice.

În cadrul pasului 4.7, problema determinării soluției pentru A . Constatăm că, în cadrul algoritmului, sunt rezolvate mai întâi pe rând problemele $P(0), P(1), \dots, P(k-1)$. Pe baza soluțiilor acestor subprobleme, se decid valorile pentru problema $P(k)$:

$$SOL_P(k) = EVALUATION(SOL_P(0), SOL_P(1), \dots, SOL_P(k-1))$$

Soluția unei probleme $P(i)$ participă la afarea soluțiilor problemelor $P(i+1), P(i+2), \dots$; de aceea, soluția unei probleme se determină doar o singură dată și se salvează, pentru a fi ulterior folosită în rezolvarea problemelor de dimensiuni mai mari. Acesta este un algoritm de programare dinamică.

Deoarece Q este cel mai mic multiplu comun al numitorilor $F[i]$, rezultă că $F'[i]$ conține doar numere întregi și $(k+1)Q$ este cel mai mic M pentru suma $S_k(n)$.

Algoritmul

Din reprezentarea algebraică de mai sus, rezultă că $M[0] = 1$, $A[0][1] = 1$, $A[0][0] = 0$.

Algoritmul este atunci:

```

ALGORITM_SUME_PUTERI
1. Read k (0 ≤ k ≤ 20)
2. genCombinations(C[k + 1][k + 1])
3. M[0] = 1, A[0][1] = 1, A[0][0] = 0
4. For(p ← 1, k; step 1) Execute
   4.1. sign = 1
   4.2. For(t ← p - 1, 0; step -1) Execute
      W[t] = simplifyFraction(sign * C[p + 1][p + 1 - t]/M[t])
      sign = sign * (-1)
   End_For
   4.3. For(t ← p, 0; step -1) Execute
      fAux = Fraction(0/1)
      For(r ← p - 1, t - 1; r ≥ 0; step -1) Execute
         fAux2 = simplifyFraction(W[r] * A[r][t])
         fAux = sumFractions(fAux, fAux2)
      End_For
      F[t] = fAux
   End_For
   4.4. fAux = simplifyFraction(W[0] * A[0][0])
   F[0] = sumFractions(F[0], fAux);
   4.5. F[p+1] = Fraction (1/1)
   4.6. M[p] = F[p]
   4.7. M[p] = (p + 1) * Q
   4.8. For(t ← p + 1, 0; step -1) Execute
      fAux2 = simplifyFraction(F[t] * Q)
      A[r][t] = Numerator(fAux)
   End_For
5. Write M[k], A[k][k+1], A[k][k], ..., A[k][1], A[k][0]
END_ALGORITM_SUME_PUTERI
```

- La pasul 2 se generează toți coeficienții binomiali până la nivelul $k+1$ (conform egalității (13) pentru calculul lui $S_k(n)$, avem nevoie de coeficienții binomiali de

$k+1$). Acest calcul se bazează tot pe un algoritm de programare dinamică cu ajutorul formulelor:

$$C[i][0] = C[i][i] = 1 \text{ pentru } 0 \leq i \leq k+1.$$

$$C[i][j] = C[i-1][j-1] + C[i-1][j] \text{ pentru } 1 \leq i \leq k+1, 1 \leq j < i$$

- La pasul 3 se inițializează valorile corespunzătoare lui $S_n(n)$ ((2) și (14)).
- La pasul 4 începe o buclă *for* în care sunt calculate secvențial valorile $A[p][0]$, $A[p][1]$, ..., $A[p][p+1]$, $M[p]$, $1 \leq p \leq k$, care sunt soluțiile problemei pentru $1, 2, \dots, k$, conform formulelor de mai sus.
- 4.1 și 4.2: fracțiile $W[]$ sunt calculate cu ajutorul formulelor (17). Metoda *simplifyFraction()* returnează fracția simplificată. Aceasta se realizează prin aflarea celui mai mare divizor comun d al numitorului și numărătorului, urmată de împărțirea acestora la d .
- Pasul 4.3 calculează fracțiile $F[]$ cu ajutorul formulei (19) și al metodei *sumFractions()* (ce returnează suma a două fracții raționale).
- Pasul 4.4. adună la $F[0]$ valoarea $W[0] \times A[0][0]$, conform primei părți a formulei (19).
- La fel, pe baza formulei (19), se inițializează fracția $F[p+1]$ cu $\frac{1}{1}$ la pasul 4.5.
- La pasul 4.6, se va calcula Q ca fiind cel mai mic multiplu comun al numitorilor fracțiilor $F[0], F[1], \dots, F[p], F[p+1]$ cu ajutorul metodei *lcmDenomin()*.
- La pasul 4.7, se calculează $M[p]$ conform (20).
- La fel, se vor calcula valorile $A[p][p+1], A[p][p], \dots, A[p][0]$ la pasul 4.8, cu ajutorul (20).

Complexitatea și corectitudinea algoritmului. Complexitatea algoritmului este $O(n^3)$ (pasul 4 conține trei bucle *for* imbricate), deci algoritmul este polinomial. El este corect, pentru că se bazează pe succesiuni corante de transformări matematice.

Într-un fel, și astăzi problema determinării sumelor de puteri k . Considerăm că, în cadrul algoritmului, sunt rezolvate mai întâi pe rând problemele $P(0), P(1), \dots, P(k-1)$. Pe baza soluțiilor acestor subprobleme, se decid valorile pentru problema $P(k)$:

$$NOI_P(i) = \text{FUNCTION}(SOL_0(i-1), SOL_1(P(1)), \dots, SOL_{k-1}(P(k-1)))$$

Soluția unei probleme $P(i)$ participă la afarea soluțiilor problemelor $P(i+1), P(i+2), \dots$; de aceea, soluția unei probleme se determină doar o singură dată și se salveză, pentru a fi ulterior folosită în rezolvarea problemelor de dimensiuni mai mari. Acesta este un algoritm de programare dinamică.

$A[k][0]$	$A[k][1]$	$A[k][2]$	$A[k][3]$	$A[k][4]$	$A[k][5]$	$A[k][6]$	$M[k]$	
0	1						1	← Soluția $k = 0$
0	1	1					2	← Soluția $k = 1$
0	1	3	2				6	← Soluția $k = 2$
0	0	1	2	1			4	← Soluția $k = 3$
0	-1	0	10	15	6		30	← Soluția $k = 4$
0	0	-1	0	5	6	2	12	← Soluția $k = 5$

Dependență. Fiecare element a pentru o putere k dată depinde de valorile a^i (daci acestea există) pentru puterile $k-1, k-2, \dots; M$ depinde de toate aceste elemente, fiind un factor de normalizare al formulei.

Programul

```
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>

const MAX_NO = 20;

using namespace std;

typedef pair<long, long> TFraction;

long cmmdc(long a, long b) {
    long r;
    while(b != 0){
        r = a % b;
        a = b;
        b = r;
    }
    return a;
}

void simplificare(TFraction f) {
    long d;
    d = cmmdc(f.first, f.second);
    if(!d){
        cout<<"Eroare!!! Impartire la 0!";
        return;
    }
    f.first /= d;
    f.second /= d;
}
```

Algoritmul lui Euclid: determinarea celui mai mare divizor comun pentru două numere naturale.

```

f.first /= d; f.second /= d;
if(f.second < 0){
    f.second *= -1;
    f.first *= -1;
}
}

fraction sumFractions(TFraction& f1, TFraction& f2){
    TFraction f;
    simplifyFraction(f1); simplifyFraction(f2);
    f.first = f1.first*f2.second + f1.second*f2.first;
    f.second = f1.second*f2.second;
    simplifyFraction(f);
    return f;
}

long lcmDenomin(TFraction fr[MAX_NO], short n){
    long m;
    short isM;
    vector<long> v;
    for(int i=0; i<n; i++){
        if(fr[i].first!=0)
            v.push_back(fr[i].second);
    }
    sort(v.begin(), v.end());
    if(v.size()<1) return 1;
    m = v[v.size()-1]; isM = 0;
    for(!isM){
        isM = 1;
        for(size_t i=1;i<v.size()&&isM;i++)
            if(m%v[i] != 0)(isM = 0; break);
        m++;
    }
    return m;
}

void genCombinations(long C[MAX_NO+2][MAX_NO+2], short n){
    int i, j;
    C[0][0] = 1;
    for(i=1; i<=n; i++){
        C[i][0] = C[i][i] = 1;
        for(j=1; j<i; j++)
            C[i][j] = C[i-1][j-1]+C[i-1][j];
    }
}

- determinarea celui mai mare divizor
  comun  $d$  al numitorului și numărătorului;
  - simplificarea cu  $d$ .

```

Determinarea celui mai mic multiplu comun al numitorilor fracțiilor $fr[]$:

- numitorii (nenuli) sunt salvați într-un vector v ;
- se începe cu cel mai mare dintre ei și acest număr se va incrementa până când devine cel mai mic multiplu comun. (O altă metodă ar fi descompunerea în factori primi a tuturor numitorilor, fiind apoi considerați toți factorii primi la puterile maxime etc.)

Coefficienții binomiali formata recursivă:

$$C_n^k = 1, \text{când } k=0 \text{ sau } n=k$$

$$C_n^k = C_{n-1}^{k-1} + C_{n-1}^k, \text{dacă } n > k, k \neq 0$$

```

int main(){
    long C[MAX_NO+2][MAX_NO+2];
    long A[MAX_NO+1][MAX_NO+2];
    long M[MAX_NO+1], Q;
    short k,p, t, r, sign;
    TFraction fr[MAX_NO+2], W[MAX_NO+1], fAux, fAux2;
    genCombinations(C, MAX_NO+1);
    ifstream in("psum.in");
    ofstream out("psum.out");
    M[0] = 1; A[0][1] = 1; A[0][0] = 0;
    for(p=1; p<=MAX_NO; p++){
        sign = 1;
        for(t=p-1; t>=0; t--){
            W[t].first = sign*C[p+1][p+1-t];
            W[t].second = M[t];
            sign *= -1;
            simplifyFraction(W[t]);
        }
        for(t=p; t>=0; t--){
            fAux.first = 0; fAux.second = 1;
            for(r=p-1; r>=t-1 && r>=0; r--){
                fAux2.first = W[r].first * A[r][t];
                fAux2.second = W[r].second;
                simplifyFraction(fAux2);
                fAux = sumFractions(fAux, fAux2);
            }
            fr[t] = fAux;
        }
        fAux2.first = W[0].first * A[0][0];
        fAux2.second = W[0].second;
        simplifyFraction(fAux2);
        fr[0] = sumFractions(fr[0], fAux2);
        fr[p+1].first = 1; fr[p+1].second = 1;

        M[p] = (p+1)*Q;
        for(t=p+1; t>=0; t--){
            fAux2.first = fr[t].first*Q;
            fAux2.second = fr[t].second;
            simplifyFraction(fAux2);
            A[p][t] = fAux2.first;
        }
    }

    while(in && !in.eof() && in>>k){
        out << M[k] << " ";
        for(t=k+1; t>=0; t--){
            out << A[k][t] << " ";
        }
    }
}

```

$S_0(n) = \sum_{i=1}^n i^0 = n = \frac{1}{1}(n+0)$

$W[i] = (-1)^{k+1-i} \frac{C_{k+1}^{k+1-i}}{M[i]}, 0 \leq i \leq k-1$

$F[0] = \sum_{i=1}^1 W[i] A[i][0] + W[0] A[0][0]$

$F[k] = \sum_{i=k-1}^{k-1} W[i] A[i][k]$

pentru orice $p, a.i. 1 \leq p \leq k$

$F[k+1] = 1$

$S_k(n) = \frac{1}{(k+1)Q} (F[k+1]n^{k+1} + F[k]n^k + F[k]n^{k-1} + \dots + F[1]n + F[0])$

$F[r] = F[r] \cdot Q, 0 \leq r \leq k+1$

```

    out << endl;
}
return 0;
}

```

Probleme propuse

1. Demonstrați formulele (3)-(7) prin inducție.
2. Citiți în *Help* metodele, structura și utilizarea tipului *std::pair* (folosit pentru reprezentarea unei fracții).
3. Tablourile bidimensionale *A[][]* și *C[][]* folosesc doar partea de sub diagonală principală. Optimizați spațiul folosit prin utilizarea în loc a doi vectori.
4. Considerăm că sumele de puteri se reprezintă în forma:

$$S_k(n) = F_{k+1}n^{k+1} + F_k n^k + \dots + F_1 n + F_0,$$

unde F_{k+1} , F_k , ..., F_0 sunt fracții simplificate; de exemplu:

$$S_0(n) = \sum_{i=1}^n i^0 = n$$

$$S_1(n) = \sum_{i=1}^n i = \frac{1}{2}n^2 + \frac{1}{2}n$$

$$S_2(n) = \sum_{i=1}^n i^2 = \frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n$$

Actualizați modelarea algebrică cu această reprezentare și scrieți programul corespunzător. Exemplu:

psum.in	psum.out
5	1/6 1/2 5/12 0 -1/12 0 0
0	1 0



München – fan bavarez la Campionatul Mondial de Fotbal (Germania, 2006)

Capitolul 6

Greedy

5 probleme complet rezolvate, 14 probleme propuse

- Fundamente
- Problema fracționară a rucsacului
- Colorarea hărții
- Calul pe tablă de șah
- Arborele parțial de cost minim (algoritmul lui Kruskal)
- Codificarea Huffman

*Când vezi că ai aceeași părere cu majoritatea,
e bine să mai reflecțezi o dată.*
Mark Twain

Fundamente

Algoritmii Greedy (engl. *greedy* = „iacom”) se caracterizează prin faptul că, la fiecare nivel, se alege mereu cel mai bun candidat posibil, după evaluarea tuturor acestora. Acești algoritmi sunt, de obicei, rapizi și furnizează o soluție relativ bună, ceea ce nu înseamnă însă că este mereu cea mai bună soluție. Acest lucru se observă ușor atunci când se aplică algoritmul pentru probleme clasice, cum ar fi: *problema discretă a rucsacului* sau *problema comisului voiajor*. Pentru aceste probleme se găsesc soluții bune, dar o soluție optimă poate fi obținută doar prin aplicarea unei tehnici de tip *Backtracking*, caz în care timpul de execuție crește însă enorm. Ambele probleme sunt *NP-complete*, atât *Greedy*, cât și *Backtracking* construiesc soluția succesiv, dar numai în cazul metodei *Backtracking* se revine mereu înapoi pe un nivel predecesor, ceea ce explică timpul foarte mare de execuție în comparație cu *Greedy*.

Forma generală a algoritmului *Greedy*:

```
ALGORITHM_Greedy( S )
    S1 ← S
    SOL ← ∅
    While (NOT STOP-Condition) Do
        x ← a local, optimal element from S1
        S1 ← S1 ∪ {x}
        If (SOL ∪ {x} satisfy conditions) Then
            SOL ← SOL ∪ {x}
        End_If
    End_While
END_ALGORITHM_Greedy( S )
```

Metoda poate fi aplicată cu succes multor probleme, printre care determinarea celor mai scurte distanțuri în grafuri (*Dijkstra*), determinarea arborelui minimal de acoperire (*Prim*, *Kruskal*), problema fracționară a rucsacului, codificarea arborilor Huffmann.

Problema 1. Problema fracționară a rucsacului



Se consideră că dispunem de un rucsac cu capacitatea M și de n obiecte, definite fiecare prin greutate și valoare, ce trebuie introduse în rucsac. Se cere o modalitate de a umple rucsacul cu obiecte, astfel încât valoarea totală să fie maximală. Este permis ca oricare obiecte și bucăți din acestea să fie introduse, iar valorile lor sunt numere reale strict pozitive.

Date de intrare. În fișierul `obiecte.in` se găsesc pe prima linie capacitatea M a rucsacului și pe următoarele linii perechi (*greutate, valoare*), pentru fiecare obiect dat, câte una pe linie. Numărul maxim de obiecte este 100, iar greutățile și valorile obiectelor pot fi considerate de tip *float*.

Date de ieșire. Scrieți în fișierul `rucsac.out` o modalitate de a umple (eventual) rucsacul, care are o valoare maximă, ca în exemplul:

<code>obiecte.in</code>	<code>rucsac.out</code>
41	Obiect 2: 23.45 600.54 - complet
12.34 123.99	Obiect 1: 12.34 123.99 - complet
23.45 600.54	Obiect 3: 12.78 90.67 - 5.21kg
12.78 90.67	
9.34 45.32	

Analiza problemei și proiectarea soluției

Obiectele date se vor sorta crescător după raportul *valoare/greutate* și apoi vor fi introduse în ordine inversă în rucsac, până când acesta se umple sau toate obiectele au fost epuizate. Ultimul obiect amplasat va fi, probabil, introdus parțial. Definim clasa *Object*, care are atributele: *g* (greutate), *v* (valoare) și *idx* (indexul corespunzător ordinii

în care obiectul a fost introdus). Implementăm constructorul și metoda *readData()* (citire și inițializare corect câmpurile *idx* ale acestora). Implementăm getter-ul *getV()* (necesar atunci când umplim rucsacul) și supraîncărcăm operatorii < (se referă la raportul *valoare/greutate* și este necesar pentru apelul metodei *sort()*), > (citirea dintr-un flux de intrare a unui obiect) << (scrierea într-un flux de ieșire a unui obiect, în forma din exemplu). Metoda *readData()* citește capacitatea M a rucsacului și obiectele din fișierul de intrare, care sunt returnate în parametrii *M*, respectiv *v*. Procesarea obiectelor citite și scrierea rezultatelor se realizează cu ajutorul metodei *processAndWrite()*. Obiectele se vor introduce succesiv în rucsac în ordine inversă și capacitatea M se micșorează corespunzător, până când, eventual, unul trebuie să fie introdus parțial; în acest caz, M devine opusul său negativ (condiție de oprire). Complexitatea algoritmului este $O(n \log n)$, datorită metodei de sortare *sort()*.

Program 1

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

class Object{
    float g, v;
    short idx;
public:
    static short cont;
    inline float const getG(){ return g; }
    bool operator<(Object&);
    friend istream& operator>>(istream&, Object&);
    friend ostream& operator<<(ostream&, Object);
};

short Object::cont = 0;

inline bool Object::operator<(Object& other){
    return v/g < other.v/other.g;
}

istream& operator>>(istream& is, Object& ob){
    is >> ob.g >> ob.v;
    ob.idx = ++Object::cont;
    return is;
}

ostream& operator<<(ostream& os, Object ob){
    os << ob.g << " " << ob.v;
    return os;
}

void readData(vector<Object> &v, ifstream &in) {
    istream in("obiecte.in");
    Object ob;
    v.clear();
    if(in && !in.eof()) in>>M;
    while(in && !in.eof() && in>>ob) {
        v.push_back(ob);
    }
}
```

```

void processAndWrite(vector<Object>& v, float& M) {
    std::ofstream out("rucsac.out");
    sort(v.begin(), v.end());
    size_t i=v.size()-1;
    while(i>=0 && M>0){
        if(M>=v[i].getG()){
            M -= v[i].getG();
            --i;
        } else{
            M = -M;
        }
    }
    for(size_t j=v.size()-1; j>i; --j){
        out << v[j] << " - complet" << std::endl;
    }
    if(i>=0 && M<0){
        out << v[i] << " - " << -M << " kg";
    }
}

int main(){
    vector<Object> v;
    float M;
    readData(v, M);
    processAndWrite(v, M);
    return 0;
}

```

Prin adăugarea în clasa *Object* a unui destructor care afișează un mesaj, vom observa că se creează mult mai multe obiecte decât sunt necesare, datorită faptului că prin manipularea cu ajutorul vectorului se creează copii ale acestora. Analizați următoarea implementare, care rezolvă problema de la pag. 107 și nu folosește de nicio copiere.

Program 2

```

#include <queue>
#include <string>

class Object{
    float g, v;
    short idx;
public:
    Object(float greutate, float valoare, short index=0)
        :g(greutate),v(valoare),idx(index){}
    inline float getG()const { return g; }

```

```
inline float getV() const { return v; }
inline short getIdx() const { return idx; }
inline bool operator<(const Object& other) const {
    return v/g < other.v/other.g;
}
};

class ObjectLessComparator{ // functor pentru operator<
public:
    inline bool operator()(const Object* pLeft,
                           const Object* pRight) const
    {return (*pLeft)<(*pRight);}
};

typedef std::priority_queue<Object*, std::vector<Object*>,
                           ObjectLessComparator>
ObjectsPriorityQueue;

std::ostream& operator<<(std::ostream& os, const Object &rObj){
    os << "Object " << rObj.getIdx() << ": ";
    os << rObj.getG() << " " << rObj.getV();
    return os;
}

void readData(ObjectsPriorityQueue &v, float& M){
    std::ifstream in("objekte.in");
    float g, val;
    short idx=0;
    if(!in.eof()) in>>M;
    while(!in.eof() && (in>>g) && (in >> val) ){
        v.push(new Object(g, val, ++idx));
    }
}

void processAndWrite(ObjectsPriorityQueue &v, float M){
    std::ofstream out("rucksack.out");
    while(!v.empty()){
        Object *pObj = v.top();
        if(M>0){
            float g = pObj->getG();
            out << (*pObj);
            M -= g;
            if(M>-0){
                out << " - vollstaendig" << std::endl;
            } else{

```

```

        out << " - " << M+g<< std::endl;
    }
}
v.pop();
delete pObj;
}
}

int main(){
    ObjectsPriorityQueue v;
    float M;
    readData(v, M);
    processAndWrite(v, M);
    return 0;
}

```

Probleme propuse

1. Implementați și o variantă C a programului.
2. În cazul varianței discrete a problemei, obiectele pot fi introduse numai complet în rucsac. În acest caz, algoritmul *Greedy* nu mai furnizează soluția optimă. Găsiți un contraexemplu.

Problema 2. Colorarea hărții

Se consideră o hartă cu n ($2 \leq n \leq 20$) țări, introdusă sub forma unei matrice $a[][]$, în care $a[i][j] = 1$ dacă țările i și j sunt vecine și $a[i][j] = 0$, dacă țările i și j nu sunt vecine.

Găsiți o modalitate de a colora harta cu cât mai puține culori, astfel încât orice două țări vecine să fie colorate diferit. Scrieți în fișierul de ieșire *culori.out* culorile determinate pentru fiecare țară, crescător de la 1 la n , ca în exemplul:

matricea	culori.out
<pre> 7 0 1 1 1 0 0 1 1 0 1 1 0 0 0 1 1 0 1 1 0 1 1 1 1 0 1 0 1 0 0 1 1 0 1 1 0 0 0 0 1 0 1 1 0 1 1 1 1 0 </pre>	<pre> 1 4 3 2 5 6 7 </pre>

Analiza problemei și proiectarea soluției

Problema se rezolvă optimal cu ajutorul metodei *Backtracking*, dar timpul de execuție este, în acest caz, exponențial. O metodă de tip *Greedy* oferă o soluție acceptabilă, dar

nu găsește mereu numărul minim de culori. Vom colora prima țară cu 0 și apoi, succesiv, toate celelalte țări, folosind pentru fiecare dintre ele cea mai mică culoare posibilă (diferită de a țărilor vecine deja „colorate”).

Program

```
#include <fstream>

int main(){
    std::ifstream fin("harta.in");
    std::ofstream fout("culori.out");
    int n, i, j, k;
    int a[50][50], col[50];
    int ok;
    fin>>n;
    for(i=0; i<n; i++)
        for(j=0; j<n; j++)
            fin>>a[i][j]; - col[0] ← 0
    col[0]=0; - căută succesiv culoarea cea mai mică posibilă pentru țara
    for(i=1; i<n; i++){ i (dacă există o țară vecină „mai mică” cu indexul k și
        j=-1; aceeași culoare, deci 1==a[k][i] && col[k]==j,
        do{ atunci ok ← 0).
            j++; ok=1;
            for(k=0; ok&&k<i; k++)
                if(1==a[k][i] && col[k]==j)
                    ok=0;
            !while(!ok);
            col[i]=j;
        }
    for(i=0; i<n; i++)
        fout<<col[i]+1<<" ";
    return 0;
}
```

Problemă propusă

Desenați convenabil graful corespunzător hărții date și găsiți o colorare a sa cu patru culori (viziile grafului sunt verificate înainte de lansarea său și nu se va ocupa de către cădă țările respective sunt vecine).

Problema 3. Calul pe tabla de șah



Scrieți un program care găsește un drum al calului pe toată tabla de șah dată de dimensiunile $m \times n$ ($4 \leq m, n \leq 100$), astfel încât fiecare câmp este pașit doar o dată.

Date de intrare. În fișierul *cal.in* se găsesc pe prima linie dimensiunile m, n ale tablei de șah și în a doua linie coordonatele de start (*linie, coloană*).

Date de ieșire. Scrieți drumul găsit în fișierul *cal.out*, astfel încât tabla de șah să fie reprezentată ca o matrice, în care poziția de start este marcată cu 1 și mutările calului sunt incrementate. Atunci când nu există soluție, se va scrie în fișierul de ieșire: „Nu există soluție”. Exemplu:

cal.in	cal.out																									
5 5 2 2	<table> <tr><td>19</td><td>12</td><td>7</td><td>2</td><td>21</td></tr> <tr><td>6</td><td>1</td><td>20</td><td>17</td><td>8</td></tr> <tr><td>11</td><td>18</td><td>13</td><td>22</td><td>3</td></tr> <tr><td>14</td><td>5</td><td>24</td><td>9</td><td>16</td></tr> <tr><td>25</td><td>10</td><td>15</td><td>4</td><td>23</td></tr> </table>	19	12	7	2	21	6	1	20	17	8	11	18	13	22	3	14	5	24	9	16	25	10	15	4	23
19	12	7	2	21																						
6	1	20	17	8																						
11	18	13	22	3																						
14	5	24	9	16																						
25	10	15	4	23																						
5 5 2 3	Nu există soluție.																									

Analiza problemei și proiectarea soluției

Vom folosi reguli regale lui Wansdorff, care a fost prezentată că autorul ei în 1803. Folosirea regulii conduce de cele mai multe ori la determinarea unei soluții și timpul de execuție este redus. Înainte de a alege mutarea următoare a calului, se vor număra, pentru fiecare câmp liber posibil, câte câmpuri posibile corespunzătoare acestuia există. Pentru evitarea „fundăturilor”, îl vom alege pe acela cu număr minim de succesiuni posibile. În funcția de verificare a unei mutări posibile, avem șanse mai mici ca în viitor să se poate ajunge aici dintr-o altă poziție.

Tabla de șah se va salva într-o matrice $a[][]$. Cele opt poziții succesoare posibile pentru un câmp vor fi definite împreună prin tablourile $dx[i]$ și $dy[i]$ (vezi programul). Dacă ne aflăm într-un câmp (x, y) , atunci cele opt câmpuri posibile unde se poate muta sunt $x + dx[i], y + dy[i]$ cu $i = 0, \dots, 7$. Metoda *nrAllowedSteps()* furnizează numărul de mutări posibile de la o poziție dată (câmpuri care se află pe tablă și nu au fost atinsă). Cel mai bun succesor pentru un câmp dat se află folosind metoda *findBestSuccessor()*. Toate câmpurile succesoare posibile sunt verificate și cel cu număr minim de câmpuri succesoare posibile este returnat. Metoda returnează *true* atunci când există cel puțin un câmp succesor și *false* atunci când nu există nici un succesor. Dacă un astfel de

succesor există, atunci coordonatele sale vor fi returnate în parametrii x și y . În programul principal, se vor determina mutările calului într-o buclă *while*, în care se apelează metoda *findBestSucc()*. Variabila *ctr* numără mutările calului.

Program

```
#include <fstream>
#include <iostream>

const short dx[8] = {-2, -2, -1, -1, 1, 1, 2, 2};
const short dy[8] = {-1, 1, -2, 2, -2, 2, -1, 1};
const short DIM_MAX = 100;

bool onTheTable(int x, int y, int m, int n){
    return
        (0<=x && x<m) &&
        (0<=y && y<n);
}

short nrAllowedSteps(short a[][DIM_MAX], int m, int n,
                     short x, short y)
{
    int nr=0;
    short xn, yn;
    for(int i=0; i<8; i++){
        xn=x+dx[i]; yn=y+dy[i];
        if(onTheTable(xn, yn, m, n)&&!a[xn][yn])nr++;
    }
    return nr;
}

bool findBestSucc(short a[][DIM_MAX], int m, int n,
                  short& x, short& y)

int aux=INT_MAX;
short xn, yn, xx, yy;
for(short i=0; i<8; i++){
    xn=x+dx[i]; yn=y+dy[i];
    if(onTheTable(xn, yn, m, n)&&
       nrAllowedSteps(a, m, n, xn, yn)<aux && !a[xn][yn]){
        aux=nrAllowedSteps(a, m, n, xn, yn);
        xx=xn; yy=yn;
    }
}
if(aux<INT_MAX){
    x=xx; y=yy;
    return true;
}
```

```
    }
    return false;
}

int main(){
    short i, j, l=3, c=3, m=4, n=4;
    short a[DIM_MAX][DIM_MAX];
    std::ifstream in("cal.in");
    std::ofstream out("cal.out");
    in>>m>>n>>l>>c; l--; c--;
    for(i=0; i<m; i++)
        for(j=0; j<n; j++)
            a[i][j]=0;
    bool flag=true;
    a[l][c]=1;
    short ctr=1;
    while(flag){
        if(findBestSucc(a, m, n, l, c))
            a[l][c]=++ctr;
        else flag=false;
    }
    if(ctr==m*n){
        for(int i=0; i<m; i++){
            out << std::endl;
            for(int j=0; j<n; j++){
                out.width(4);
                out << a[i][j] << " ";
            }
        }
    }else{
        out<<"Nu există solutie.";
    }
    return 0;
}
```

Problema propusă

Scriți o variantă recursivă pentru problema dată.

Problema 4. Arborele parțial de cost minim (algoritmul lui Kruskal)



Pentru a înțelege mai bine problema, vom face mai întâi o recapitulare a cîtorva concepte de bază din teoria grafurilor.

Definiția 1. Un **graf** este o pereche $G = (V, E)$, în care V este o mulțime finită și nevidă de vîrfuri (engl. *vertices* = „noduri”), iar E este o mulțime de perechi din $V \times V$, numite muchii (engl. *edges*).

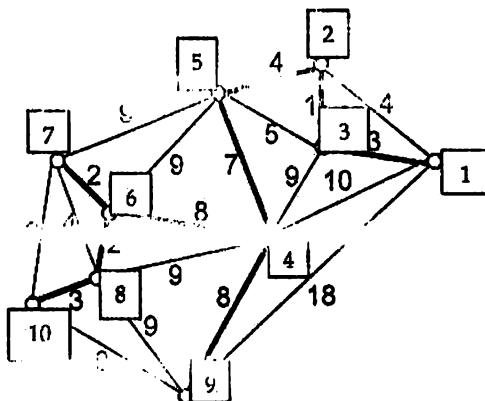
Definiția 2. O funcție $c : E \rightarrow \mathbb{R}$, prin care fiecărei muchii i se atribuie un număr real pozitiv, se numește **funcție de cost a grafului G** .

Definiția 3. Un **graf parțial** al grafului $G = (V, E)$ este un graf $H = (V, F)$, cu proprietatea că $F \subseteq E$ (H are toate vîrfurile lui G , dar nu neapărat toate muchiile lui G).

Definiția 4. Fie $H = (V, F)$ un **graf parțial** al grafului $G = (V, E)$. Prin **costul** grafului parțial H se înțelege suma costurilor muchiilor sale: $c(H) = \sum_{e \in F} c(e)$.

Definiția 5. Un **graf parțial $H = (V, F)$** al grafului $G = (V, E)$, care conține toate vîrfurile lui G și este arbore, se numește **arbore parțial** al lui G .

Definiția 6. Un arbore parțial având costul minim peste mulțimea arborilor parțiali ai unui graf dat G se numește **arbore parțial de cost minim**.



Arbore parțial de cost minim

Se dă un graf, împreună cu o funcție de cost asociată și se cere determinarea arborelui parțial de cost minim.

Date de intrare. În fișierul *kruskal.in* se găsește un graf neorientat dat prin matricea costurilor astfel: pe prima linie numărul n al vârfurilor ($1 \leq n \leq 200$), iar pe următoarele n linii matricea pătratică reprezentând costurile muchiilor, $c[i][j] > 0$ costul muchiei (i, j) , $c[i][j] = 0$, dacă nu există o muchie (i, j) . Valorile $c[i][j]$ sunt numere reale.

Date de ieșire. Scrieți rezultatele în fișierul de ieșire *kruskal.out*, căte o linie cu perechi de muchii și costul lor, urmate de costul total al arborelui parțial minim determinat, ca în exemplu:

<i>kruskal.in</i>	<i>kruskal.out</i>
10	(2, 3) -> 1
0 4 3 10 0 0 0 0 18 0	(6, 7) -> 2
4 0 1 0 5 0 0 0 0 0	(6, 8) -> 2
3 1 0 9 5 0 0 0 0 0	(1, 3) -> 3
10 0 9 0 7 8 0 9 8 0	(8, 10) -> 3
0 4 5 7 0 9 9 0 0 0	(2, 5) -> 5
0 0 0 8 9 0 2 2 0 0	(4, 5) -> 7
0 0 0 0 9 2 0 4 0 6	(4, 6) -> 8
0 0 0 9 0 2 4 0 9 3	(4, 9) -> 8
18 0 0 8 0 0 0 9 0 9	-----
0 0 0 0 0 6 3 9 0	Cost: 39

Analiza problemei și proiectarea soluției

Cei mai cunoscuți algoritmi pentru determinarea arborelui parțial de cost minim sunt cei ai lui Kruskal și Prim. Vom prezenta în continuare algoritmul lui Kruskal, propus de matematicianul american Josep' Bernard Kruskal (n. 29.01.1928, New York) în *Proceedings of the American Mathematical Society*, 1956.

Algoritmul lui Kruskal (1956). Fie date graful $G = (V, E)$ cu n vârfuri și o funcție de cost asociată $c : E \rightarrow \mathbb{R}$. Se începe cu graful parțial $H = (V, \emptyset)$, care are n componente conexe (este o pădure). Acesta se va transforma succesiv într-un arbore parțial cu n vârfuri și $n - 1$ muchii. Mai întâi se introduce în H muchia de cost minim din G , pas prin care se obține un graf parțial cu $n - 1$ componente conexe. La fiecare pas, se va alege muchia de cost minim din E , care nu formează un ciclu în H , se introduce în H și se elimină din E . Aceasta înseamnă că alegem muchia de cost minim, ale cărei vârfuri se află în componente diferite din H . Prin această operație, numărul componentelor conexe se micșorează cu 1. Algoritmul se termină când au fost alese $n - 1$ muchii. Compl. $O(|E| \log |V|)$. În pseudocod:

ALGORITM_KRUSKAL(Graf G)

```

SortCrescător( $e_1, e_2, \dots, e_m$ )
For ( $i=1, n$ ; step 1) Execute
     $C[i] \leftarrow i$                                 // inițializarea componentelor
End_For
 $cost \leftarrow 0, H \leftarrow \emptyset$ 
 $muchiiSelectate \leftarrow 0$ 
While ( $muchiiSelectate < n - 1$ ) Do
     $(u, v) \leftarrow urmMuchie(E)$ 
    While ( $C[u] \neq C[v]$ ) Do
         $(u, v) \leftarrow urmMuchie (E)$ 
    End_While
     $H.add((u, v))$                                 // adaugă muchia  $(u, v)$  la  $H$ 
     $cost \leftarrow cost + c(u, v)$ 
     $muchiiSelectate \leftarrow muchiiSelectate + 1$ 
     $max \leftarrow \max(C[u], C[v])$ 
     $min \leftarrow \min(C[u], C[v])$ 
    For ( $i=1, n$ ; step 1) Execute           // unirea a două componente
        If( $C[i] = max$ ) Then  $C[i] \leftarrow min$  End_If
    End_For
End_While
return  $H, cost$ 

```

END ALGORITM_KRUSKAL(Graf G)

Algoritmul conduce la obținerea unui arbore parțial minim și aceasta se poate

Pentru salvarea grafului și a arborelui parțial vom folosi o structură de tip `std::multimap`, ce conține perechi în care pe prima poziție este costul și pe a doua este un număr întreg reprezentând codificarea unei muchii. Avantajul este că muchiile se vor afla în *multimap* mereu în ordinea crescătoare a costurilor lor. Folosim *multimap*, și nu *map*, pentru că o cheie (costul unei muchii) poate să apară repetat. Dacă n este numărul vârfurilor grafului, atunci muchia (i, j) se va reprezenta prin $i * n + j$ (pentru a regăsi i și j pe baza unei codificări k : $i \leftarrow k \text{ div } n, j \leftarrow k \text{ mod } n$). Deoarece matricea costurilor este simetrică față de diagonala principală, vom salva în graf doar partea superioară a acestieia.

```

in >> aux;
if(i<j && aux)
    E.insert(TPair(aux, i * n + j)); // inserăză în E perechea (aux, i * n + j)

```

Urmăriți în program implementarea algoritmului lui Kruskal și utilizarea tipurilor `std::multimap` și `std::pair`, a iteratorilor și a metodelor `insert` și `erase`.

Program

```

#include <iostream>
#include <vector>
#include <map>

using namespace std;

typedef pair<double, int> TPair;

int main(){
    ifstream in("kruskal.in");
    ofstream out("kruskal.out");
    multimap<double, int> E, H;
    double aux;
    short n;
    if(in && !in.eof() && in>>n){
        for(short i=0; in && !in.eof() && i<n; i++)
            for(short j=0; in && !in.eof() && j<n; j++){
                in >> aux;
                if(i<j && aux)
                    E.insert(TPair(aux, i*n+j));
            }
    }

    H.clear();
    vector<short> C;
    for(short i=0; i<n; i++) C.push_back(i);
    while(H.size() < n-1){
        short idx = 0;
        short uu, vv;
        multimap<double, int>::iterator it;
        it = E.begin();
        uu = (it->second)/n; vv = (it->second)%n; // „decodificarea” lui it->second
        while(C[uu]==C[vv]){
            it++;
            uu = it->second/n;
            vv = it->second%n;
        };
    }
}

```

```

H.insert(TPair(it->first, it->second));      // inserează muchie în H
E.erase(it);                                    // șterge muchie din E
short mi = min(C[uu], C[vv]);
short ma = max(C[uu], C[vv]);
for(short i=0; i<n; i++)
    if(C[i]==ma) C[i]=mi;
}

multimap<double, int>::const_iterator it;
double cost = 0;
for(it=H.begin(); it!=H.end(); it++){
    out << "(" << it->second/n+1 << ", " << it->second%n+1
        << ") -> " << it->first << endl;
    cost += it->first;
}
out << "-----" << endl
    << "Cost: " << cost;

return 0;
}

```

Probleme propuse

1. Scrieți pe hârtie pașii și structurile intermediare pentru exemplul dat.
2. Modularizați programul prin împărțirea în metode *citireDate()*, *constrArboreKruskal()*, *afisareArbore()* și apoi folosiți-le în programul principal.
3. Scrieți o implementare C pentru problema.
4. Un alt algoritm cunoscut pentru determinarea arborelui parțial minim, tot de tip *Greedy*, este *algoritmul lui Prim*. Scrieți algoritmul și programul corespunzătoare.

Problema 5. Codificarea Huffman



Presupunem că avem un text cu 262 de caractere și ne dorim stocarea lui într-o formă cât mai redusă. Textul conține șase caractere (*q*, *r*, *t*, *x*, *y*, *z*), iar *q* și *t* sunt 100 de ori, *q* de 17 ori, *r* de două ori, *x* de 58 de ori, *y* de 80 de ori și *z* de cinci ori.

Modul ușual de a reprezenta informațiile în calculator este cel folosind biții 0 și 1. Ne interesează aşadar găsirea unei modalități de reprezentare a caracterelor folosind un *cod binar*, astfel încât fiecare

caracter este reprezentat cu ajutorul unei secvențe binare unice. În cazul nostru, dacă vom folosi un *cod binar de lungime fixă*, avem nevoie de minimum 3 biți pentru fiecare dintre cele șase caractere. Exemplu: $p = 000$, $q = 001$, $r = 010$, $x = 011$, $y = 100$, $z = 101$. În total, avem nevoie de $262 \cdot 3 = 786$ biți.

O alternativă la codul de lungime fixă este *codul de lungime variabilă*, care poate conduce la îmbunătățiri semnificative. Acest cod urmărește alocarea unor secvențe binare cât mai scurte caracterelor cel mai des folosite. Exemplu pentru textul nostru: $p = 0$, $q = 1101$, $r = 11000$, $x = 111$, $y = 10$, $z = 11001$. Numărul de biți necesari este: $100 \times 1 + 17 \times 4 + 2 \times 5 + 58 \times 3 + 80 \times 2 + 5 \times 5 = 537$, iar aceasta reprezintă 68,3% din 786, adică am obținut o reducere de circa 31,7% în raport cu codificarea de lungime fixă.

*Codurile fix și variabil pentru un text de 262 de caractere,
ce conține caracterele p, q, r, x, y, z, cu frecvențele indicate*

	<i>p</i>	<i>q</i>	<i>r</i>	<i>x</i>	<i>y</i>	<i>z</i>	Total biți
Frecvență	100	17	2	58	80	5	-
Cod fix	000	001	010	011	100	101	786
Cod variabil	0	1101	11000	111	10	11001	537

Această codificare este validă, deoarece nici un cuvânt binar asociat unui caracter nu este prefix al cuvântului binar asociat altui caracter. O astfel de codificare se numește *codificare prefix și simplifică atât codificarea* (deci compactarea), cât și decodificarea. De exemplu, cuvântul binar 1011000110010111 se decodifică unic în *yrpzx*.

O astfel de codificare prefix a fost propusă de către David Albert Huffman (1923-1999) în septembrie 1952 (*Proceedings of the I.R.E.*, pp. 1098-1102); se folosește un arbore binar ale căruia frunze sunt caracterele date și interpretăm fiecare cod binar pentru un caracter ca fiind drumul de la rădăcină până la caracterul respectiv, unde 0 reprezintă deplasare pe fiul stâng și 1 deplasare pe fiul drept. Arborele atașat unei pagini din revista *Proceedings of the I.R.E.* din 1952 arată următoarea reprezentare:

arbore binar de căutare deoarece frunzele nu trebuie să fie ordonate și nodurile interne nu conțin cheii pentru caractere.

Dacă notăm cu C alfabetul caracterelor, atunci arboarele are $|C|$ frunze, câte una pentru fiecare litere și $|C| - 1$ noduri interne. Dacă notăm cu $f(c)$ frecvența unui caracter și cu $h(c)$ adâncimea frunzei în arbore (numărul de biți în codificare), atunci numărul total de caractere necesare este $\sum_{c \in C} f(c) \cdot h(c)$ și se mai numește *costul arborelui*.

Algoritmul lui Huffman este unul de tip *Greedy*: se începe cu o mulțime $|C|$ de frunze și se realizează $|C| - 1$ operații de fusionare, în final obținându-se un singur arbore. La fiecare pas, se vor alege doi arbori de cost minim și se vor înlocui cu arborele obținut prin fuzionarea celor doi arbori. Prin fuzionarea a doi arbori, A_1 și A_2 ,

Înțelegem crearea unui nou arbore binar, în care cei doi arbori A_1 și A_2 sunt fiilii stângi, respectiv drept ai rădăcinii.

Scrieți un program ce determină codificarea Huffman pentru un text dat, cu frecvențele caracterelor date.

Date de intrare. În fișierul *huffman.in* se găsesc perechile (caracter, frecvență), câte una pe linie. Caracterele pot fi litere ale alfabetului englez, iar frecvențele sunt numere naturale între 1 și 1000.

Date de ieșire. Scrieți în fișierul *huffman.out* codificările pentru fiecare caracter, ca în exemplul:

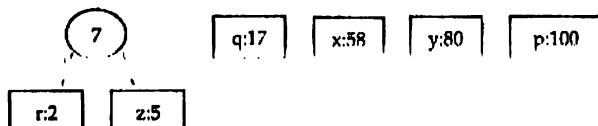
<i>huffman.in</i>	<i>huffman.out</i>
p 100	p 0
q 17	y 10
r 2	r 11000
x 58	z 11001
y 80	q 1101
z 5	x 111

Analiza problemei și proiectarea soluției

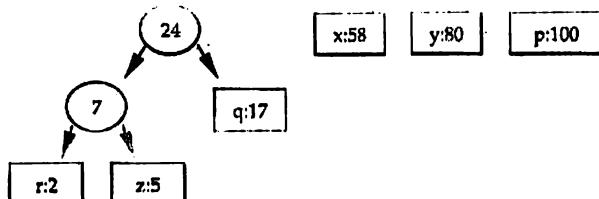
Evoluția arborilor pentru exemplul dat este prezentată mai jos. Se începe cu șase noduri frunză:



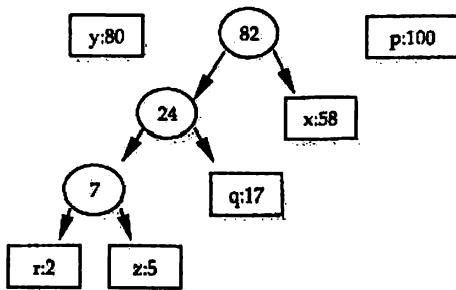
Se unesc arborii de cost minim și se obțin următorii cinci arbori:



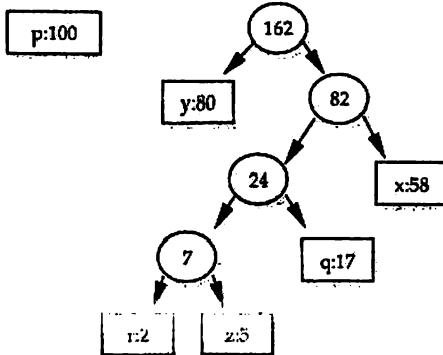
Prin fuzionarea arborilor de costuri 7 și 17, se obține:



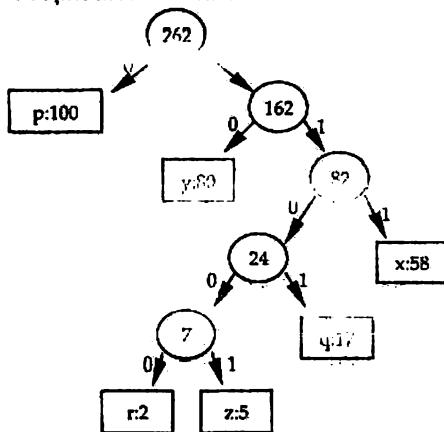
Arborii de costuri minime sunt cei cu 24 și 58 în rădăcină:



La următorul pas, ultimii doi arbori au costurile 100 și 162:



Prin unirea lor, se obține arborele final:



în care am etichetat muchiile cu valorile 0 și 1. Acest arbore binar complet conține codificarea Huffman, ea fiind pentru fiecare caracter sirul etichetărilor muchiilor drumului de la rădăcină la frunza corespunzătoare.

Algoritmul în pseudocod:

```

ALGORITM_Huffman(C)
    M ← ArbořiFrnzăInițiali(C)
    While (M are mai mult de un element) Do
        A1, A2 ← ArboriiCuCostMinim(M),
        Sterge A1, A2 din M
        A ← ArboreFuzionat(A1, A2)
        M.add(A)
    End_While
    return Codificări(M)
END_ALGORITM_Huffman(C)

```

Complexitatea algoritmului este $O(n \log n)$, datorită operației căutare a arborilor de cost minim. Algoritmul lui Huffman conduce la o codificare optimă, datorită proprietății de substructură optimă a subproblemelor. Demonstrații pentru aceasta se găsesc, de exemplu, în [Cor00] și [Knu99].

Vom scrie clasa *Node*, care va conține ca atribută un caracter, costul și cei doi arbori filii. Vom supraîncărca operatorul „<”, deoarece ne propunem să salvăm mulțimea de arbori într-o structură de tip *std::multimap*, care îi va păstra ordonații și la fiecare pas de procesare îi vom înlocui pe primii doi cu cel obținut prin fuzionarea lor, folosind constructorul cu parametri doi arbori. Metoda menținută în *Node* va scrie în fișierul de ieșire codificările Huffman ale arborelui.

Program

```

#include <iomanip>
#include <vector>
#include <fstream>

class Node{
    unsigned int _cost;
    char _ch;
    Node *_nLeft, *_nRight;
public:
    Node(unsigned cost, char ch):

```

```

    _cost(cost), _ch(ch), _nLeft(NULL), _nRight(NULL)()
Node(Node* pLeftChild,Node* pRightChild)
    :_cost(pLeftChild->_cost+pRightChild->_cost),
     _ch(0),_nLeft(pLeftChild),_nRight(pRightChild){}
~Node()();
inline unsigned int getCost() const {return _cost; }

inline bool operator <(const Node &other) const
    return _cost < other._cost;
}
void writeCosts(std::ofstream& out,
                 std::vector<bool> &rPathPrefix);
};

void Node::writeCosts(std::ofstream& out,
                      std::vector<bool> &rPathPrefix)
{
if(_nLeft==NULL || _nRight==NULL) { //nod frunză
    out << _ch << "t";
    for(std::vector<bool>::const_iterator it=rPathPrefix.begin();
        it!=rPathPrefix.end();it++) {
        out << (*it ? 1:0);
    }
    out<<std::endl;
}
else { //nod interior
    rPathPrefix.push_back(false);
    _nLeft->writeCosts(out,rPathPrefix);
    rPathPrefix[rPathPrefix.size()-1] = true;
    _nRight->writeCosts(out, rPathPrefix);
    rPathPrefix.pop_back();
}
}

int main(){
    std::ifstream in("huffman.in");

    unsigned cost; char ch;
    std::multimap<int, Node*> s;
    while(in && !in.eof() && in>>ch>>cost){
        s.insert(std::pair<int, Node*>(cost, new Node(cost, ch)));
    }

    if(s.empty()) {
        return 0;
    }

    while(s.size()>1){

```

```

std::multimap<int, Node*>::iterator it = s.begin();
Node *pFirstNode = it->second;
s.erase(it);
it = s.begin();
it = s.begin();
Node *pSecondNode = it->second;
s.erase(it);
Node *pNewNode = new Node(pFirstNode, pSecondNode);
s.insert(std::pair<int,Node*>(pNewNode->getCost(), pNewNode));
}

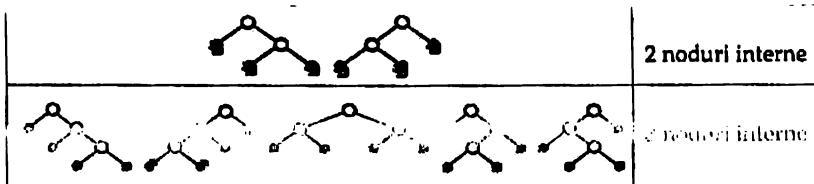
s.begin()->second->writeCosts(std::ofstream("huffman.out"),
                                std::vector<bool>());
return 0;
}

```

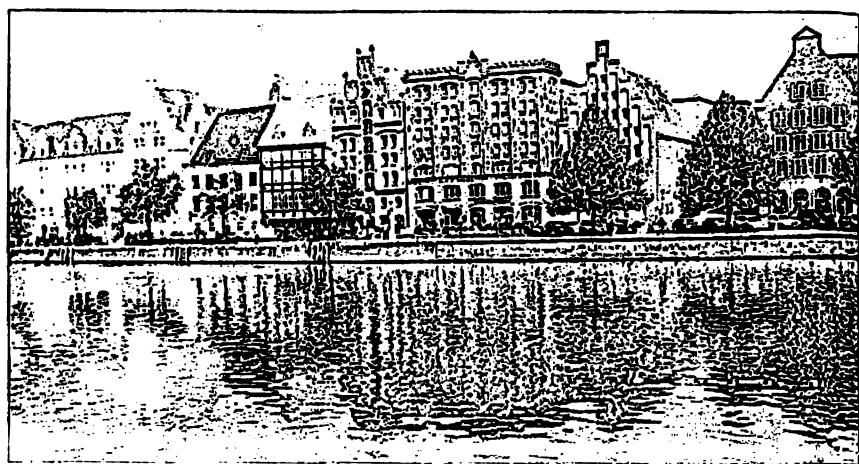
Probleme propuse

- Desenați arborii Huffman pentru codificarea cuvintelor **BANANA**, **ABRACADABRA** și **MISSISSIPPI**.
- Demonstrați că algoritmul Huffman conduce la o codificare optimă.
- Scrieți o implementare C a algoritmului.
- Extindeți programul astfel încât să furnizeze și procentul de reducere al codificării Huffman în raport cu codificarea de lungime fixă.
- Modificați programul astfel încât la intrare să primească un fișier text și să se compacteze acest fișier folosind codificarea Huffman.
- Demonstrați că numărul tuturor arborilor binari compleți cu n noduri interne este

$$\text{al } n\text{-lea număr din sirul lui Catalan: } C_n = \frac{1}{n+1} C_{2n}^n.$$



Desenați cei 14 arbori binari compleți cu 4 noduri interne.



Case oglindite in Lillebeck

Capitolul 7

Problema ordonării datelor

- Descrierea problemei
- Domeniul problemei. Definiții
- *POD și PODI* sunt NP-complete
- Algoritmi pentru *POD* și *PODI*
- Algoritmii *Random (RAN)*
- Algoritmii *Exact (EX)*
- Algoritmii *Greedy Min (GM)*
- Algoritmii *Greedy Min Simplified (GMS)*
- Algoritmii de mărginire inferioară (*LB*)
- Detalii de implementare
- Program (*STL*, parametri în linia de comandă)
- Interpretarea rezultatelor
- Probleme propuse

*Dacă vrei să construiești un vapor, să nu începi prin
a-i trimite pe oameni după lemne, cuie, unele, sfuri
și alte materiale. Învață-i întâi să tânjească după
marea îndepărtată, nesfârșită.*

Antoine de Saint-Exupéry

Descrierea problemei

Se consideră o mulțime de cuvinte (mesaje) ce urmează să fie transmise pe un canal astfel încât ordinea de transmitere este irelevantă. Problema este determinarea secvenței optime care minimizează numărul total de tranziții pe canal.

În 1994, Stan și Burleson au prezentat metoda *bus-invert*: cuvintele pot fi inversate și apoi transmise pentru a reduce numărul total de transmisii ([Bur94]).

Prin *inversarea unui cuvânt* înțelegem operația prin care fiecare bit din cuvânt primește valoarea complementului său ($0 \rightarrow 1$, $1 \rightarrow 0$). Exemplu: pentru $w = 10110001$, $w_{\text{inversat}} = 01001110$. Vom încerca în continuare să combinăm cele două paradigmă într-o singură, adică vom permite cuvintelor să fie complementate, reordonate și apoi transmise. (Vom demonstra că această problemă – *Problema Ordonării Datelor cu Inversiune* [PODI] este NP-completă.) Aria de aplicații practice ale acestei probleme devine din ce în ce mai largă, de la testarea circuitelor până la transmiterea pachetelor în cadrul internetului și de la *low-power design* până la implicații în biologie.

Domeniul problemei. Definiții

Definiția 1. Distanța Hamming. Dacă cuvântul w_r este transmis urmat imediat de w_s , atunci numărul total de tranziții este dat de numărul de biți care se schimbă. Acesta este:

$$d(w_r, w_s) = \sum_{j=1}^n w_{rj} \oplus w_{sj}, \quad (1)$$

cunoscută, de asemenea, ca *distanța Hamming* dintre w_r și w_s . Aici, w_{rj} este al j -lea bit din reprezentarea lui w_r și \oplus este operația *SAU-LXCLUSIV*:

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

De exemplu, $d(10001110, 10110111) = 4$.

În cazul în care dispunem de mai multe cuvinte, reordonarea cuvintelor poate schimba semnificativ numărul total de tranziții.

În continuare, vom nota cu \bar{w} complementul cuvântului w . Exemplu: $\overline{10110} = 01001$.

Propoziția 1. Fie k un număr natural pozitiv. Distanța Hamming definită pe spațiul cuvintelor de lungime k este o metrică pe spațiul total al cuvintelor, deoarece satisface următoarele:

1. $d(x, y) = 0 \Leftrightarrow x = y$ oricare ar fi două cuvinte de lungime k ;
2. $d(x, y) = d(y, x)$ (simetrie);
3. $d(x, y) \leq d(x, z) + d(z, y)$ (inegalitatea triunghiului).

Demonstrație:

1. Din faptul că $d(x, y) = 0$, deducem:

$$\sum_{j=0}^k d(x_j, y_j) = 0 \Leftrightarrow d(x_j, y_j) = 0, \forall j = 1, k \Leftrightarrow x_j = y_j, \forall j = 1, k \Leftrightarrow x = y \quad (x_j \text{ și } y_j \text{ sunt bițiile din compoziția cuvintelor } x \text{ și } y).$$

2. Egalitatea este evidentă datorită simetriei operației \oplus .

3. Presupunem că x , y și z , cele trei cuvinte, au forma binară: $x_1...x_k$, $y_1...y_k$ și $z_1...z_k$. Afirmația este adeverată pentru cuvinte de lungime 1, dacă vom considera bițiile a , b , c , atunci tabelul pentru expresiile $d(a, b)$ și $d(a, c) + d(c, b)$ este:

$$d(a, b) = 1 - \delta(f_a, f_b)$$

a	b	c	$d(a, b)$	$d(a, c) + d(c, b)$
0	0	0	0	0
0	0	1	0	2
0	1	0	1	1
0	1	1	1	1
1	0	0	1	1
1	0	1	1	1
1	1	0	0	2
1	1	1	0	0

Din acest tabel observăm că inegalitatea triunghiului este satisfăcută pentru cuvinte de lungime 1 și, deci, însumând biții corespunzători, deducem că este adevărată pentru cuvinte de orice lungime. □

Definiția 2. Numărul total de tranzitii. Considerându-se cuvintele w_1, w_2, \dots, w_n cu semnificația de mai sus, o permutare δ (o funcție bijectivă cu domeniul $\{w_1, w_2, \dots, w_n\}$ și codomeniul tot $\{w_1, w_2, \dots, w_n\}$) a lor și o asignare ρ (o funcție cu domeniul $\{1, 2, \dots, n\}$ și codomeniul $\{0,1\}$) care specifică pentru fiecare cuvânt dacă se va considera complementat sau necomplementat, definim numărul total al tranzitilor ca fiind valoarea:

$$N_T = \sum_{j=1}^{n-1} d(\delta(w_{\rho(j)}), \delta(w_{\rho(j+1)})) \quad (2)$$

Definiția 3. Matricea de adiacență. Pentru o instanță a problemei (unde n este numărul de cuvinte și k lungimea fiecărui), definim matricea de adiacență $A_{n \times n}$ unde $A(i, j) = d(w_i, w_j)$.

Exemplu: Considerăm $n = 3, k = 5$ și cuvintele:

$$w_1 = 00011$$

$$w_2 = 10110$$

$$w_3 = 01011$$

$$A = \begin{pmatrix} 0 & 3 & 1 \\ 3 & 0 & 4 \\ 1 & 4 & 0 \end{pmatrix}$$

$$(d(w_1, w_1) = 0, d(w_1, w_2) = 3, d(w_1, w_3) = 1, \dots).$$

Referitor la această definiție pentru matrice, deducem:

1. Matricea este simetrică față de diagonala principală. Datorită faptului că operația de SAU-EXCLUSIV este comutativă deducem că distanța Hamming este comutativă, adică:

$$d(x \oplus y, z) = d(x, z) + d(y, z)$$

✓

2. Elementele de pe diagonala principală sunt nule. Din definiția distanței Hamming deducem că:

$$d(w_i, w_i) = 0 \Rightarrow a_{ii} = 0 \quad (5)$$

3. Dacă inversăm toate cuvintele, matricea de adiacență este aceeași, ca și în cazul când acestea sunt în forma inițială:

$$d(w_i, w_j) = d(\overline{w_i}, \overline{w_j}) \quad (6)$$

4. Dacă lungimea cuvintelor este k , atunci este adevărată următoarea relație:

$$d(w_i, \overline{w_j}) = d(\overline{w_i}, w_j) = k - d(w_i, w_j) = k - a_{ij} \quad (7)$$

5. Dacă vom considera cuvintele în ordinea inițială, atunci numărul total de tranziții se poate obține însumând elementele de pe diagonalele vecine cu diagonala principală (deasupra și dedesubtul acesteia).

Exemplul 1. Considerăm $n = 3, k = 12$ și următoarele cuvinte:

$$w_1 = 010100111000$$

$$w_2 = 001010100110$$

$$w_3 = 001111011011$$

$$A = \begin{pmatrix} 0 & 8 & 7 \\ 8 & 0 & 7 \\ 7 & 7 & 0 \end{pmatrix}$$

Dacă vom considera cuvintele în ordinea inițială și neinversate, atunci numărul total de tranziții va fi:

$$N_T = a_{12} + a_{23} = 15$$

Dacă inversăm al doilea cuvânt, obținem:

$$N_T = (k - a_{12}) + (k - a_{23}) = (12 - 8) + (12 - 7) = 9.$$

acea ce înseamnă că numărul total de tranziții s-a diminuat cu 40%.

Exemplul 2. Considerăm $n = 7, k = 15$ și următoarele cuvinte:

$$w_1 = 001010100110110$$

$$w_2 = 000000011111110$$

$$w_3 = 011111011011000$$

$$w_4 = 000010011110111$$

$$w_5 = 000001111011011$$

$$w_6 = 000110010001100$$

$$w_7 = 000000010000111$$

$$A = \begin{pmatrix} 0 & 5 & 7 & 5 & 9 & 8 & 8 \\ 5 & 0 & 8 & 4 & 6 & 7 & 5 \\ 7 & 8 & 0 & 6 & 8 & 5 & 6 \\ 5 & 4 & 6 & 0 & 6 & 7 & 7 \\ 9 & 6 & 6 & 6 & 0 & 9 & 5 \\ 8 & 7 & 7 & 7 & 9 & 0 & 4 \\ 8 & 5 & 7 & 7 & 5 & 4 & 0 \end{pmatrix}$$

Dacă vom considera cuvintele în ordinea inițială și neinversate, numărul total de tranziții va fi:

$$N_T = \sum_{i=1}^6 a_{i,i+1} = 38$$

Dacă vom schimba doar ordinea cuvintelor $w_1, w_2, w_4, w_3, w_5, w_7, w_6$ și le vom păstra neinversate, obținem:

$$N_T = 30$$

ceea ce ar însemna o reducere de 26,67%.

Exemplul 3. Considerăm $n = 8, k = 13$ și următoarele cuvinte:

$$\begin{aligned}w_1 &= 1111010111001 \\w_2 &= 1010100110110 \\w_3 &= 0000001111110 \\w_4 &= 1111011010010 \\w_5 &= 0001111011011 \\w_6 &= 0110010001100 \\w_7 &= 0001010001110 \\w_8 &= 1000110110100\end{aligned}$$

$$A = \begin{pmatrix} 0 & 8 & 9 & 5 & 7 & 6 & 8 & 7 \\ 8 & 0 & 5 & 7 & 9 & 8 & 8 & 3 \\ 9 & 5 & 0 & 8 & 6 & 7 & 5 & 6 \\ 5 & 7 & 8 & 0 & 6 & 7 & 7 & 8 \\ 7 & 9 & 6 & 6 & 0 & 9 & 5 & 8 \\ 6 & 8 & 7 & 7 & 9 & 0 & 4 & 7 \\ 8 & 8 & 5 & 7 & 5 & 4 & 0 & 7 \\ 7 & 3 & 6 & 8 & 8 & 7 & 7 & 0 \end{pmatrix}$$

Dacă vom considera cuvintele în ordinea inițială și neinversate, numărul total de tranziții va fi:

$$N_T = \sum_{i=1}^7 a_{i,i+1} = 47$$

Dacă vom considera cuvintele reordonate și unele inverse ale acestora astfel:

$$w_1 \rightarrow \bar{w}_3 \rightarrow w_4 \rightarrow \bar{w}_8 \rightarrow \bar{w}_2 \rightarrow w_5 \rightarrow w_6 \rightarrow w_7$$

obținem următorul număr de tranziții:

$$N_T = (13 - a_{13}) + (13 - a_{34}) + (13 - a_{48}) + a_{82} + (13 - a_{25}) + a_{56} + a_{67} = 29$$

iar reducerea obținută este de 38,30%.

În aceste exemple, obținerea că reordonarea și recitarea unui singur cuvânt poate duce la reduceri mari de resurse. Dacă pentru aceste cazuri, în care dimensiunile n și k sunt foarte mici, folosirea unui algoritm deterministic pentru aflarea optimului este posibilă, atunci când valorile cresc, un astfel de algoritm nu are utilitate practică. Soluția ar fi căutarea unor metode care să ne furnizeze valori căt mai apropiate de valorile optime.

Definiția 4. POD (Problema ordonării datelor). Găsiți o permutare σ a șirului de cuvinte w_1, w_2, \dots, w_n , astfel încât numărul total de tranzitii

$$N_T = \sum_{j=1}^{n-1} d(w_{\sigma(j)}, w_{\sigma(j+1)}) \quad (8)$$

să fie minimizat.

Definiția 5. PODI (Problema ordonării datelor cu inversiune). Găsiți o permutare σ a șirului de cuvinte w_1, w_2, \dots, w_n și o asignare δ astfel încât numărul total de tranzitii

$$N_T = \sum_{j=1}^{n-1} d(\delta(w_{\sigma(j)}), \delta(w_{\sigma(j+1)})) \quad (9)$$

să fie minimizat.

POD și PODI sunt NP-complete

În [Mur95] se demonstrează că POD este NP-completă și au fost propuse o serie de scheme pentru găsirea unor soluții acceptabile: arbore parțial dublu (*Double Spanning Tree – DST*), arbore parțial minim cu potrivire maximă (*Minimum Spanning Tree Maximum Matching – ST-MM*) și *Greedy* (care din punct de vedere empiric a fost găsită drept cea mai bună).

Casa NP reprezintă cea mai interesantă clasă de probleme. Una dintre explicații ar fi aceea că, dacă se găsește un algoritm polinomial ce rezolvă o problemă NP-completă, atunci există către un algoritm polinomial pentru rezolvarea oricărei probleme NP-complete. Datorită v. scării pe care o acoperă problemele NP-complete studiate până în prezent fără să se fi făcut vreun mic pas în direcția rezolvării vreunei dintre ele, ar fi uluiitor să se descopere vreodată rezolvarea unei astfel de probleme în timp polinomial.

În cadrul acestui capitol, vom prezenta o metodă de rezolvare a POD-ului, furnizând o demonstrație a inaccesibilității sale. Pentru un inginer, de exemplu, este deseori mai indicat dacă, relativ la o problemă din această clasă, încearcă să găsească un algoritm de aproximare decât să caute un algoritm exact. Există un număr foarte mare de probleme obișnuite și interesante care la prima vedere nu par să fie mai dificile, decât o sortare sau o căutare în graiuri și care, totuși, sunt NP-complete. Problemele NP-complete apar în cele mai diverse domenii: logică, grafuri, aritmetică, proiectarea rețelelor, mulțimi și partiții, memorări și căutări, planificări, programare matematică, algebră și teoria numerelor, jocuri, teoria limbajelor și a automatelor, optimizarea programelor etc.

PODI se transformă într-o problemă de decizie după logica următoare.

Instanță. Se consideră o mulțime de n cuvinte w_1, w_2, \dots, w_n , fiecare de dimensiune k : $w_i \in \{0, 1\}^k$ și un număr pozitiv M .

Întrebare. Există permutare σ a șirului de cuvinte w_1, w_2, \dots, w_n și o asignare δ a celor n cuvinte, astfel încât

$$\sum_{j=1}^{n-1} d(\delta(w_{\sigma(j)}), \delta(w_{\sigma(j+1)})) \leq M ? \quad (10)$$

($\delta(w_i) = w_i$ sau $\delta(w_i) = \overline{w_i}$).

Stim că *POD* este NP-completă și, intuitiv, se deduce că și *PODI* este cel puțin tot atât de grea, deoarece are un grad mai mare de libertate privind plasarea obiectelor.

Teoremă. *PODI* este NP-completă.

Demonstrație. Vom transforma *POD* în *PODI*. *POD* are ca problemă de decizie următoarea formă:

Instanță. Se consideră o mulțime de N cuvinte W_1, W_2, \dots, W_N , fiecare de dimensiune K : $W_i \in \{0, 1\}^K$, și un număr pozitiv L .

Întrebare. Există permutare σ a celor N cuvinte date, astfel încât

$$\sum_{j=1}^{n-1} d(W_{\sigma(j)}, W_{\sigma(j+1)}) \leq L ? \quad (11)$$

Considerându-se o instanță pentru *POD*, generăm următoarea instanță pentru *PODI*. Fie $n = N$ și $M = L$. Pentru fiecare i , construim w_i concatenând KN zerouri după W_i , adică $w_i = W_i \cup \dots \cup KN$ (KN zerouri). Astfel, $k = K+KN$ și timpul de transformare este polinomial. Atunci există o permutare σ a cuvintelor W_i , astfel încât relația (11) este satisfăcută dacă și numai dacă există o permutare a cuvintelor w_i , astfel încât relația (10) este satisfăcută. \square

Algoritmi pentru *POD* și *PODI*

Presupunem că avem la dispoziție tipurile abstracte *Permutation* și *BitString*, care permit manipularea permutărilor, respectiv a sirurilor de biți. Vom descrie în continuare căruia algoritmii în pseudocod pentru *PODI*, problema fiind o extindere a problemei *POD*. Pentru o instanță a problemei, adică n cuvinte de lungime k , *PODI* necesită returnarea unei permutări de ordin n (repräsentând ordinea de transmitere a cuvintelor) și a unui șir de biți tot de lungime n (repräsentând asignările cuvintelor). Algoritmii descriși vor fi *Random POD* și *PODI*, *Exact POD* și *PODI*, două metode *Greedy* pentru ambele probleme și un algoritm de mărginire inferioară (engl. *lower-bound*), folosind arborele parțial minim de acoperire.

Considerăm $n = 8, k = 9$ și cuvintele:

$w_1 = 010110011$
 $w_2 = 101110000$
 $w_3 = 001000001$
 $w_4 = 111010001$
 $w_5 = 111011111$
 $w_6 = 000101111$
 $w_7 = 100010001$
 $w_8 = 001001111$

Matricea de adiacență $A_{\text{EX89}} =$	$\begin{pmatrix} 0 & 5 & 5 & 4 & 5 & 4 & 4 & 6 \\ 5 & 0 & 4 & 3 & 6 & 7 & 3 & 7 \\ 5 & 4 & 0 & 3 & 6 & 5 & 3 & 3 \\ 4 & 3 & 3 & 0 & 3 & 8 & 2 & 6 \\ 5 & 6 & 6 & 3 & 0 & 5 & 5 & 3 \\ 4 & 7 & 5 & 8 & 5 & 0 & 6 & 2 \\ 4 & 3 & 3 & 2 & 5 & 6 & 0 & 6 \\ 6 & 7 & 3 & 6 & 3 & 2 & 6 & 0 \end{pmatrix}$
---	--

Vom numi această instanță a problemei EX89 și ne vom referi la ea pentru a exemplifica soluțiile algoritmilor prezentați în continuare. Costul total pentru cazul când cuvintele sunt transmise în ordinea inițială și neinversate este 32.

Algoritmii Random (RAN)

Algoritmul *Random* pentru *POD* va genera o permutare aleatorie și o va returna, împreună cu costul atașat. Cel pentru *PODI* va genera o permutare și un sir de biți reprezentând asignarea.

ALGORITM_RAN_POD($n, k, w_1 \dots w_n$)

```
p ← random Permutation
return p, cost(p)
```

END_ALGORITM_RAN_POD($n, k, w_1 \dots w_n$)

ALGORITM_RAN_PODI($n, k, w_1 \dots w_n$)

```
p ← random Permutation
b ← random BitString
return p, b, cost(p, b)
```

END_ALGORITM_RAN_PODI($n, k, w_1 \dots w_n$)

De exemplu, pentru permutarea aleatorie $(7, 8, 4, 2, 3, 5, 6, 1)$, costul *POD* pentru *EX89* este 34. Pentru permutarea $(7, 1, 3, 8, 5, 2, 4, 6)$ și asignarea $(0, 1, 0, 1, 1, 1, 0, 0)$, costul *PODI* este 38.

Algoritmii *Exact* (*EX*)

Algoritmul *Exact* pentru *POD* generează toate permutările și o reține pe prima, care conduce la un cost minim. Cel pentru *PODI* generează toate perechile posibile permutare-asignare și o rețin pe prima cu costul minim.

Algoritmul *Exact* pentru *POD* va genera o permutare aleatorie și o va returna, împreună cu costul atașat. Cel pentru *PODI* va genera o permutare și un sir de biți reprezentând asignarea.

```
ALGORITM_EXACT_POD( $n, k, w_1 \dots w_n$ )
     $p \leftarrow$  random Permutation( $n$ )
    For (all Permutations( $n$ )  $r$ ) Execute
        If( $\text{cost}(r) < \text{cost}(p)$ )  $p \leftarrow r$ 
    End_For
    return  $p, \text{cost}(p)$ 
END_ALGORITM_EXACT_POD( $n, k, w_1 \dots w_n$ )
```

```
ALGORITM_EXACT_PODI( $n, k, w_1 \dots w_n$ )
     $p \leftarrow$  random Permutation( $n$ )
     $b \leftarrow$  random BitString( $n$ )
    For (all BitStrings( $n$ )  $i$ ) Execute
        For (all BitStrings( $n$ )  $bs$ ) Execute
            If( $\text{cost}(r, bs) < \text{cost}(p, b)$ ) Then
                 $p \leftarrow i$ 
                 $b \leftarrow bs$ 
            End_If End_For End_For
        return  $p, b, \text{cost}(p, b)$ 
END_ALGORITM_EXACT_PODI( $n, k, w_1 \dots w_n$ )
```

Pentru exemplul EX89, costul minim *POD* este 21 și se obține cu permutarea (1, 6, 8, 3, 2, 7, 4, 5). Costul minim *PODI* este 16 și se obține cu permutarea (1, 8, 2, 6, 4, 7, 3, 5), iar asignarea este (0, 1, 0, 1, 0, 0, 0, 1). Complexitatea algoritmului *EXACT_POD* este $O(n!)$; pentru *EXACT_PODI* este $O(n! \cdot 2^n)$.

Algoritmii Greedy Min (GM)

Greedy Min funcționează pentru ambele probleme, *POD* și *PODI*. El funcționează după cum urmează:

- calculează costul pentru toate perechile de cuvinte distincte și selectează perechea cu cost minim;
- această pereche cu cost minim este secvența de început, formată din două cuvinte;
- construiește progresiv secvența, adăugând la fiecare pas cel mai convenabil cuvânt care nu a fost încă adăugat. Acest cuvânt poate fi adăugat la sfârșitul sau la începutul secvenței, după cum costul obținut este minimal. În cazul *PODI* se adaugă, bineînțeles, și libertatea de inversare a cuvântului adăugat.

Complexitatea algoritmului este polinomială $O(n^2)$ pentru ambele probleme. Pentru exemplul EX89, permutarea obținută pentru *POD* este (5, 1, 6, 8, 3, 4, 7, 2) și conduce la costul 22. Pentru *PODI* se obțin permutarea (5, 3, 7, 4, 6, 2, 8, 1) și asignarea (1, 0, 0, 0, 1, 0, 1, 0) și conduc la un cost total 16.

Algoritmii Greedy Min Simplified (GMS)

Aceștia funcționează, de asemenea, pentru ambele probleme, *POD* și *PODI*. Diferența este că, la fiecare pas, adăugarea nouului cuvânt (cu posibilitatea de a se seta și inversă, în cazul *PODI*) este permisă doar la un capăt, al secvenței. Explicația este că se economisește timp prin testarea numai la un capăt și performanțele în practică pentru

transmitere a cuvintelor (prin intermediul internetului, de exemplu), timpul este un factor important. Complexitatea este, de asemenea, polinomială $O(n^2)$.

Pentru exemplul EX89, permutarea obținută pentru *POD* este (4, 7, 2, 3, 8, 6, 1, 5) și conduce la costul total 23. Pentru cazul *PODI*, se obțin permutarea (4, 6, 8, 1, 2, 7, 3, 5) și asignarea (0, 1, 1, 0, 1, 1, 1, 0), care conduc la costul 19.

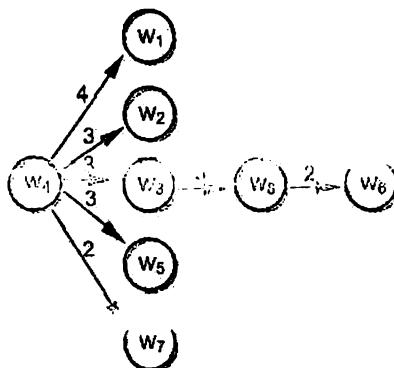
Algoritmii de mărginire inferioară (LB)

Problemele *POD* și *PODI* se pot modela natural cu ajutorul teoriei grafurilor. Pentru *POD* se construiește graful complet cu vârfurile reprezentând cele n cuvinte și muchiile având drept costuri distanța dintre cuvintele reprezentând capetele sale. Conform secțiunii teoretice de mai sus, distanța dintre două cuvinte este aceeași dacă

amândouă sunt în aceeași stare de inversare (ambele inversate sau ambele în forma inițială). De asemenea, dacă ele sunt în stări diferite, distanța este aceeași pentru ambele posibilități. În cazul *PODI*, se va construi un multigraf: între două vârfuri diferite rutem avea și două muchii, reprezentând amândouă posibilitățile (ambele aceeași fază sau faze diferite), atunci când cele două costuri rezultate sunt diferite. Problemele *POD* și *PODI* cer determinarea unor drumuri hamiltoniene de cost minim în graf, respectiv multigraf, care sunt probleme *NP*-complete. Un arbore minimal de acoperire pe acest graf (respectiv multigraf) va avea un cost mai mic decât drumul hamiltonian de cost minim. Nu putem folosi un astfel de arbore ca soluție pentru problema dată, dar rezultatul se poate utiliza pentru a măsura deviațiile de la minim realizate de algoritmii *greedy* prezenți mai sus.

Pentru determinarea arborelui parțial de cost minim există doi algoritmi: *Prim* (folosește vârfurile) și *Kruskal* (folosește muchiile). Vom implementa algoritmul *Kruskal*, prezentat în capitolul precedent, pentru a furniza limite inferioare ale costurilor.

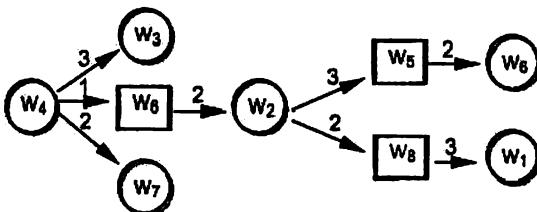
Pentru *POD*, exemplul EX89, se va obține arborele din figură ($(4, 7) \rightarrow 2 \mid (6, 8) \rightarrow 2 \mid (2, 4) \rightarrow 3 \mid (3, 4) \rightarrow 3 \mid (3, 8) \rightarrow 3 \mid (4, 5) \rightarrow 3 \mid (1, 4) \rightarrow 4$).



Arbore parțial de cost minim pentru *POD*, EX89

Costul obținut este așadar 20.

Pentru *PODI* se va obține arborele: $(4, 6) \rightarrow 1 \mid (2, 6) \rightarrow 2 \mid (2, 8) \rightarrow 2 \mid (4, 7) \rightarrow 2 \mid (1, 8) \rightarrow 3 \mid (2, 5) \rightarrow 3 \mid (3, 4) \rightarrow 3$, care are costul 16.



Arbore parțial de cost minim pentru PODI, EX89

Complexitatea algoritmului este polinomială $O(n^2)$.

Detalii de implementare

Ca de obicei, vom lăsa deoparte în cadrul metodelor din program validarea corectitudinii datelor de intrare. Vom presupune că toate sunt corecte și nu vom testa, de exemplu, dacă există cuvinte de lungimi diferite, dacă avem mai puțin de două cuvinte sau dacă o permutare dată este într-adevăr corectă. Acest lucru ne va permite să ne concentrăm pe funcționarea algoritmilor și a structurilor alese. Vom implementa toți algoritmii prezentați mai sus pentru ambele probleme, *POD* și *PODI*. Vom defini tipurile *BiString* (șir de biți – *vector<bool>*), *Permutation* (permutare), *TPair* (perechi necesare pentru determinarea arborelui parțial minim), *MMMI* (*multimap*, tot pentru algoritmul Kruskal – salvarea grafului și a arborelui).

Metoda *generateWords(n, k, w)* va genera aleatoriu n cuvinte binare, fiecare de lungime k , și le va returna în parametrul prin referință w . Metoda *hammingDist(w1, w2)* returnează distanța Hamming dintre două cuvinte date (numărul de biți corelați care corespund în cele două cuvinte). *constructMatrix(w, n)* construiește matricea de adiacență pe baza sirului de cuvinte w ($a[i][j]$ va conține numărul de biți corelați diferenți). *getCostPOD()* și *getCostPODI()* returnează numărul total de tranziții pentru instanțe ale celor două probleme (șir de cuvinte, matricea de adiacență, permutare, pentru *PODI*, adițional, și o asignare bs).

Metodele *getRandomCostPOD()* și *getRandomCostPODI()* foloizează aleatoriu sortuj ale celor două probleme și returnează costurile asociate lor. Remarcați utilizarea algoritmului *random_shuffle()* din header-ul *<algorithm>*, care amestecă elementele unui container situat între iteratorii dați ca parametru.

getExactPOD() determină o soluție optimă pentru problema *POD* și o returnează în al treilea parametru – permutarea p . Inițial, p este permutarea identică, toate permutările se vor genera lexicografic și permutarea optimă va fi mereu actualizată. Remarcați utilizarea algoritmului *next_permutation()*, care generează următoarea

permutare în ordine lexicografică a permutterii reprezentate de iteratorii-parametru. Metoda returnează costul optimul. `getExactPODI()` determină soluția optimă pentru o instanță a `PODI` și o returnează în ultimii doi parametri – permutterea `p`, asignarea `bs`. Pentru fiecare permutterare se vor genera toate asignările posibile (de la 000...00 la 111...11) și se va reține optimul actual. Pentru a determina succesorul unui sir de biți, scriem funcția `next_bitstring()`, care transformă parametrul în succesorul său, dacă acesta există, caz în care returnează `true`. Exemplu: $\text{succesor}(110111011) = 110111100$. Dacă succesorul nu există, `next_bitstring()` returnează `false`. Succesorul unui sir de biți se obține prin transformarea în 1 a primului bit 0, considerat de la dreapta spre stânga și transformarea în 0 a tuturor bițiilor succesorii (toți sunt 1!).

Metoda `getLB()` returnează limita inferioară a unei instanțe și se poate aplica ambelor probleme prin setarea corespunzătoare a celui de-al treilea parametru `isPODI`. Dacă se cere rezolvarea `PODI`, atunci se vor introduce în multigraful inițial și costurile muchiilor pentru cuvintele cu asignări diferite ($k - a[i][j]$). Implementarea este foarte asemănătoare cu cea prezentată detaliat în capitolul 6, *Greedy*, la problema 4 (Kruskal).

`getGMS()` rezolvă problemele `POD` și `PODI` folosind algoritmul *Greedy Min Simplified*, prezentat mai sus. Dacă se cere rezolvarea `POD`, atunci vectorul `b` va conține numai elemente `false` – nici un cuvânt nu se va inversa. În continuare, se caută perechea de cuvinte ($i0, j0$) care au distanță minimă, pentru cazul `PODI` se verifică și valorile $k - a[i][j]$, care se va adăuga în permutterea `p`. Pentru cazul `PODI`, trebuie actualizată și asignarea `b`:

```

p.push_back(i0);
p.push_back(j0);
if(isPODI){
    b.push_back(false);
    if(a[i0][j0]==cost) b.push_back(false);
    else b.push_back(true);
}

```

Cât timp permutterea `p` curentă nu are încă n elemente (`p.size() < n`), se caută următorul cuvânt care va fi introdus. Acest cuvânt (indice `next`) nu trebuie să se afle încă în `p` (`find(p.begin(), p.end(), i) == p.end()`) și să formeze cost minim cu ultimul element din `p` ($j0$). Remarcați folosirea algoritmului `find()` pus la dispoziție de *STL*. Pentru cazul `PODI`, se verifică și valoarea $k - a[j0][i]$. Valoarea `next` găsită se va adăuga lui `p` și, în cazul `PODI`, se va actualiza și asignarea `b` în concordanță cu ultimul element introdus în `p` și ultimul element din `b`.

Metoda *Greedy Min* este implementată prin `getGM()`. Deoarece ea permite operarea la ambele capete ale secvenței curente, vom folosi liste auxiliare `pAux` și `bAux` pentru a reprezenta permutterea și asignarea. Vom reține de această dată ambele capete ale

sevenței în $i0$ și $j0$, la fiecare pas se va adăuga cuvântul cel mai convenabil, încă neadăugat, ținând cont și dacă se cere rezolvarea PODI. La sfârșit, cele două liste se vor copia în p și b prin:

```
for(size_t i=0; i<n; i++){
    p.push_back(pAux.front()); pAux.pop_front();
    if(isPODI){
        b.push_back(bAux.front());
        bAux.pop_front();
    }
}
```

Remarcați utilizarea metodelor `pop_front()`, `push_front()`, `front()`, caracteristice listelor.

Ne propunem aplicarea tuturor algoritmilor implementați pentru diferite instanțe aleatorii ale problemei. Pentru aceasta vom folosi șase argumente în linia de comandă: $n1, n2, sn, k1, k2, sk$, cu semnificația că se vor genera și analiza succesiv instanțe pentru valori ale lui n între $n1$ și $n2$, cu pasul sn , și pentru fiecare valori ale lui k între $k1$ și $k2$, cu pasul sk :

```
for(n=n1; n<=n2; n+=sn)
    for(k=k1; k<=k2; k+=sk)... // aplică toți algoritmii ...
```

Dacă numele executabilului este `podpodi.exe`, atunci se va executa comanda:

```
podpodi 5 9 1 25 65 20
```

Rezultatele se vor scrie în fișierul `report.txt`, în următoarea formă:

POD							PODI						
n	k	sum	avr	med	sd	cv	sum	avr	med	sd	cv		
5	25	48	40	41	40	37	61	35	38	35	34		
5	45	90	79	84	79	79	91	76	79	76	76		
5	65	134	117	120	117	117	134	116	118	116	115		
6	25	66	55	55	55	55	58	47	47	47	47		
6	45	114	95	98	95	95	121	101	101	101	101		
6	65	158	146	155	146	142	167	136	139	136	136		
...													

Urmărîți implementarea în programul principal. Deoarece ne propunem apeluri la tuturor algoritmilor, deci și a celor exacti, timpul de execuție poate fi considerabil. De aceea, valorile pentru n și k trebuie să fie reduse.

Program (STL, parametri în linia de comandă)

```
#include <vector>
#include <algorithm>
#include <map>
#include <list>
#include <fstream>

using namespace std;

typedef vector<bool> BitString;
typedef vector<unsigned> Permutation;

typedef pair<unsigned, unsigned> TPair;
typedef multimap<unsigned, unsigned> MMUU;

bool generateWords(unsigned n, unsigned k,
                    vector<BitString>& w){
    for(unsigned i=0; i<n; i++){
        BitString wAux; short moneda;
        for(unsigned j=0; j<k; j++){
            moneda = (int)(2*((double)rand()/(RAND_MAX+1)));
            wAux.push_back(moneda ? true : false);
        }
        w.push_back(wAux);
    }
    return true;
}

int hammingDist(BitString w1, BitString w2){
    if(w1.size() != w2.size()) return -1;
    int t=0;
    for(size_t i=0; i<w1.size(); i++)
        if(w1[i] != w2[i]) t++;
    return t;
}

void constructMatrix(vector<BitString>& w,
                     vector<vector<unsigned> >& a){
    unsigned n = w.size();
    unsigned k = w[0].size();
    for(unsigned i=0; i<n; i++)
        for(unsigned j=0; j<n; j++)
            a[i][j] = hammingDist(w[i], w[j]);
}
```

```

unsigned getCostPOD(vector<BitString>& w,
                     vector<vector<unsigned> >& a,
                     Permutation& p){
    unsigned t = 0;
    for(size_t i = 1; i<w.size(); i++)
        t += a[p[i-1]][p[i]];
    return t;
}

unsigned getCostPODI(unsigned n, unsigned k,
                     vector<vector<unsigned> >& a,
                     Permutation& p, BitString& bs){
    unsigned t = 0;
    if(n <=1 || p.size()!=n || bs.size()!=n)
        return 0;
    for(unsigned i = 1; i<n; i++)
        t += (bs[i]==bs[i-1])?a[p[i-1]][p[i]]:k-a[p[i-1]][p[i]];
    return t;
}

unsigned getRandomCostPOD(unsigned n, unsigned k,
                           vector<vector<unsigned> >& a,
                           Permutation& p){
    for(unsigned i=0; i<n; i++)p.push_back(i);
    random_shuffle(p.begin(), p.end());
    return getCostPOD(n, k, a, p);
}

unsigned getRandomCostPODI(unsigned n, unsigned k,
                           vector<vector<unsigned> >& a,
                           Permutation& p, BitString& b){

    p.clear(); b.clear();
    for(unsigned i=0; i<n; i++){
        p.push_back(i);
        b.push_back((int)(2*((double)rand()/(RAND_MAX+1))));
    }
    random_shuffle(p.begin(), p.end());
    cout << " ";
    getCostPODI(n, k, a, p, b);
}

unsigned getExactPOD(unsigned n, unsigned k,
                     vector<vector<unsigned> >& a,
                     Permutation& p){
    unsigned t=getRandomCostPOD(n, k, a, p);

```

```
Permutation r;
p.clear();
for(unsigned i=0; i < n; i++) r.push_back(i);
while(next_permutation(r.begin(), r.end())){
    unsigned aux = getCostPOD(n, k, a, r);
    if(aux<t){
        t=aux;
        p=r;
    }
}
return t;
}

bool next_bitstring(BitString& bs){
    int idx = bs.size()-1;
    while(idx>=0 && bs.at(idx)){
        bs[idx--]=false;
    }
    if(idx>=0){
        bs[idx] = true;
        return true;
    }
    return false;
}

unsigned getExactPODI(unsigned n, unsigned k,
                      vector<vector<unsigned> >& a,
                      Permutation& p, BitString& bs){
    unsigned r=getRandomCostPOD(n, k, a, p, bs);
    Permutation r;
    p.clear();
    for(unsigned i=0; i < n; i++) r.push_back(i);
    while(next_permutation(r.begin(), r.end())){
        BitString bs2;
        for(unsigned i=0; i < n; i++) bs2.push_back(false);
        do{
            unsigned aux = getCostPODI(n, k, a, r, bs2);
            if(aux<t){
                t=aux;
                p=r;
                bs=bs2;
            }
        } while(next_bitstring(bs2));
    }
    return t;
}
```

```

unsigned getLB(unsigned n, unsigned k,
              vector<vector<unsigned>>& a,
              bool isPODI, MMUU& H){
    H.clear();
    MMUU E;
    for(unsigned i=0; i<n-1; i++)
        for(unsigned j=0; j<n; j++){
            E.insert(TPair(a[i][j], i*n+j));
            if (isPODI) E.insert(TPair(k-a[i][j], i*n+j));
        }
    vector<unsigned> C;
    unsigned cost = 0;
    for(unsigned i=0; i<n; i++) C.push_back(i);
    while(H.size()<n-1){
        unsigned uu, vv;
        MMUU::iterator it = E.begin();
        uu = (it->second)/n; vv = (it->second)%n;
        while(C[uu]==C[vv]){
            it++;
            uu = it->second/n;
            vv = it->second%n;
        };
        H.insert(TPair(it->first, it->second));
        cost += it->first;
        E.erase(it);
        unsigned mi = min(C[uu], C[vv]);
        unsigned ma = max(C[uu], C[vv]);
        for(i=uu; i<n-1; i++)
            if(C[i]<=ma) C[i]=mi;
    }
    return cost;
}

unsigned getGMS(unsigned n, unsigned k,
                vector<vector<unsigned>>& a,
                bool isPODI,
                Permutation& p, BitString& b){
    p.clear(); b.clear();
    if(!isPODI)
        for(unsigned i=0; i<n; i++) b.push_back(false);
    unsigned i0=0, j0=1;
    unsigned cost = a[i0][j0];
    for(unsigned i=0; i<n-1; i++){
        for(unsigned j=i+1; j<n; j++){
            if(a[i][j]<cost){
                i0=i; j0=j;
                cost = a[i0][j0];
            }
        }
    }
}

```

```
    cost = a[i0][j0];
}
if(isPODI && k-a[i][j]<cost){
    i0=i; j0=j;
    cost = k-a[i][j];
}
}
p.push_back(i0);
p.push_back(j0);
if(isPODI){
    b.push_back(false);
    if(a[i0][j0]==cost) b.push_back(false);
    else b.push_back(true);
}
while(p.size()<n){
    unsigned next, aux = k+1;
    for(unsigned i=0; i<n; i++){
        if(find(p.begin(), p.end(), i)==p.end()&&a[j0][i]<aux){
            if(a[j0][i]<aux){
                aux = a[j0][i]; next = i;
            }
            if(isPODI && k-a[j0][i]<aux){
                aux = k-a[j0][i]; next = i;
            }
        }
    }
    p.push_back(next);
    cost += aux;
    if(isPODI){
        bool bLast = b[b.size()-1];
        if(aux==a[j0][next]) b.push_back(bLast);
        else b.push_back(!bLast);
    }
    i0=next;
}
return cost;
}

unsigned getGM(unsigned n, unsigned k,
               vector<vector<unsigned>> a,
               bool isPODI,
               Permutation& p, BitString& b){
p.clear(); b.clear();
list<unsigned> pAux;
list<bool> bAux;
if(!isPODI)
    for(unsigned i=0; i<n; i++) b.push_back(false);
unsigned i0=0, j0=1;
```

```

unsigned cost = a[i0][j0];
for(unsigned i=0; i<n-1; i++){
    for(unsigned j=i+1; j<n; j++){
        if(a[i][j]<cost){
            i0=i; j0=j;
            cost = a[i0][j0];
        };
        if(isPODI && k-a[i][j]<cost){
            i0=i; j0=j;
            cost = k-a[i][j];
        }
    }
    pAux.push_back(i0);
    pAux.push_back(j0);
    if(isPODI){
        bAux.push_back(false);
        if(a[i0][j0]==cost) bAux.push_back(false);
        else bAux.push_back(true);
    }
}
while(pAux.size()<n){
    unsigned next, aux = k+1;
    for(unsigned i=0; i<n; i++){
        if(find(pAux.begin(), pAux.end(), i)==pAux.end())){
            if(a[j0][i]<aux){
                aux = a[j0][i]; next = i;
            };
            if(a[i0][i]<aux){
                aux = a[i0][i]; next = i;
            };
            if(isPODI && k-a[i0][i]<aux){
                aux = k-a[i0][i]; next = i;
            }
        }
    }
    cost += aux;
    if(a[i0][next]==aux || (isPODI && k-a[i0][next]==aux)){
        pAux.push_back(next);
        if(isPODI){
            bool bLast = bAux.back();
            if(bLast)
                bAux.pop_back();
            bAux.push_back(true);
        }
    }
}

```

```
        if(aux==a[j0][next]) bAux.push_back(bLast);
        else bAux.push_back(!bLast);
    }
    j0 = next;
} else {
    pAux.push_front(next);
    if(isPODI){
        bool bFront = bAux.front();
        if(aux==a[i0][next]) bAux.push_front(bFront);
        else bAux.push_front(!bFront);
    }
    i0=next;
}

for(unsigned i=0; i<n; i++){
    p.push_back(pAux.front()); pAux.pop_front();
    if(isPODI){
        b.push_back(bAux.front());
        bAux.pop_front();
    }
}

return cost;
}

int main(int argc, char ** argv) {
ofstream out("report.txt");
if(argc != 7) {
    out << "Numar incorrect de argumente pentru program!";
    return 0;
}

unsigned n1, n2, sn, k1, k2, sk;
unsigned n, k;
n1 = atoi(argv[1]);
n2 = atoi(argv[2]);
sn = atoi(argv[3]);
k1 = atoi(argv[4]);
k2 = atoi(argv[5]);
sk = atoi(argv[6]);

short D=5;
out.width(4*D); out << "POD";
out.width(5*D); out << "PODI" << endl;
out.width(D); out<<"n"; out.width(D); out<<"k";
```

```

out << " | ";
out.width(D); out<<"RAN"; out.width(D); out<<"EX";
out.width(D); out<<"GMS";
out.width(D); out<<"GM"; out.width(D); out<<"LB";
out << " | ";
out.width(D); out<<"RAN"; out.width(D); out<<"EX";
out.width(D); out<<"GMS";
out.width(D); out<<"GM"; out.width(D); out<<"LB";
out << endl;

for(n=n1; n<=n2; n+=sn)
  for(k=k1; k<=k2; k+=sk) {

    vector<BitString> w;
    Permutation p;
    BitString b;
    MMUU H;
    vector<vector<unsigned> > a(n, vector<unsigned>(n, 0));
    generateWords(n, k, w);
    constructMatrix(w, a);

    out.width(D); out << n;
    out.width(D); out << k;
    out << " | ";
    out.width(D); out << getRandomCostPOD(n, k, a, p);
    out.width(D); out << getExactPOD(n, k, a, p);
    out.width(D); out << getGMS(n, k, a, false, p, b);
    out.width(D); out << getPOD(n, k, a, false, p, b);
    out.width(D); out << getLB(n, k, a, false, H);
    out << " | ";
    out.width(D); out << getRandomCostPODI(n, k, a, p, b);
    out.width(D); out << getExactPODI(n, k, a, p, b);
    out.width(D); out << getGMS(n, k, a, true, p, b);
    out.width(D); out << getPODI(n, k, a, true, p, b);
    out.width(D); out << getLB(n, k, a, true, H);
    out << endl;
  }
  return 0;
}

```

Interpretarea rezultatelor

Tabelul următor conține rezultatele obținute de algoritmii *RAN*, *EX*, *GMS*, *GM* și *LB* pentru instanțe mici ale problemei, folosind programul anterior.

Rezultate POD/PODI pentru instanțe de dimensiune redusă

n	k	POD					PODI				
		RAN	EX	GMS	GM	LB	RAN	EX	GMS	GM	LB
5	25	48	31	41	40	37	53	30	38	35	34
5	45	84	75	77	75	75	96	74	77	74	74
5	65	130	116	119	119	113	144	111	112	111	111
6	25	69	50	52	50	50	63	44	48	44	44
6	45	98	91	96	93	91	106	82	84	82	82
6	65	156	118	148	136	134	156	113	137	134	130
7	25	68	58	60	59	56	80	56	59	56	55
7	45	143	116	116	116	115	133	104	114	104	102
7	65	204	180	185	180	179	179	158	166	159	155
8	25	81	73	75	74	69	85	64	70	65	62
8	45	149	141	146	142	140	153	127	135	127	125
8	65	235	204	208	208	202	217	185	195	189	183

Remarcăm că, pentru aceeași instanță a problemei, rezultatele algoritmului EX pentru PODI sunt, aşa cum era de așteptat, mai bune decât cele pentru POD (datorită gradului mai mare de libertate adăugat acesteia). Algoritmul de mărginire inferioară, LB, furnizează rezultate mai mici, dar destul de apropiate de cele exacte. Îmbunătățirea adusă de PODI în comparație cu POD este cu atât mai mare, cu cât lungimea k a cuvintelor este mai mare.

Pentru a compara cei doi algoritmi Greedy pe instanțe mari ale problemelor, putem să le apelăm în aceeași manieră; de exemplu, pentru n , din mulțimea {1250, 1450, ..., 2050}, în număr de k cu același k din mulțimea {5, 10, 15, 20}. Algoritmii aplicati fiecărei probleme: Random (RAN), Greedy Min (GM), Greedy Min Simplified (GMS) și mărginire inferioară (LB). Vom introduce delta (Δ), ce reprezintă deviația lui GMS în raport cu GM. Pentru un studiu mai exact al comportării celor doi algoritmi, pentru fiecare perioadă (n, k) au fost aplicate 15 rulări și valorile medii sunt completeate

În cadrul algoritmului GMS, acesta este mai rapid decât GM (aproape de două ori, când se consideră matricea de adiacență construită).

Rezultate POD

Input Data		POD				
n	k	RAN	GM	GMS	Δ	LB
1250	15	9341	2480	2478	-2	2102
1250	315	196999	163523	163535	+12	160951
1250	615	383842	337871	337861	-10	334211
1450	15	10868	2766	2755	-11	2342
1450	315	228241	189220	189217	-3	186318
1450	615	445842	391117	391146	+29	386907
1650	15	12363	3031	3041	+10	2582
1650	315	259686	214691	214698	+7	211377
1650	615	507187	444244	444233	-11	439538
1850	15	13915	3315	3314	-1	2890
1850	315	291342	239950	239984	+34	236325
1850	615	568827	497320	497371	+51	492047
2050	15	15277	3570	3588	+18	3025
2050	315	322854	265499	265533	+34	261566
2050	615	630214	550324	550376	+52	544575

Remarcăm că algoritmii *Greedy* furnizează rezultate foarte bune comparativ cu cel *Random* și că rezultatele produse de algoritmul *LB* sunt, de asemenea, destul de apropiate. Sunt 6 cazuri din 15 în care *GMS* furnizează, în medie, rezultate mai bune decât *GM* și, pentru celelalte, diferența Δ este neglijabilă, raportat la lungimea cuvintelor.

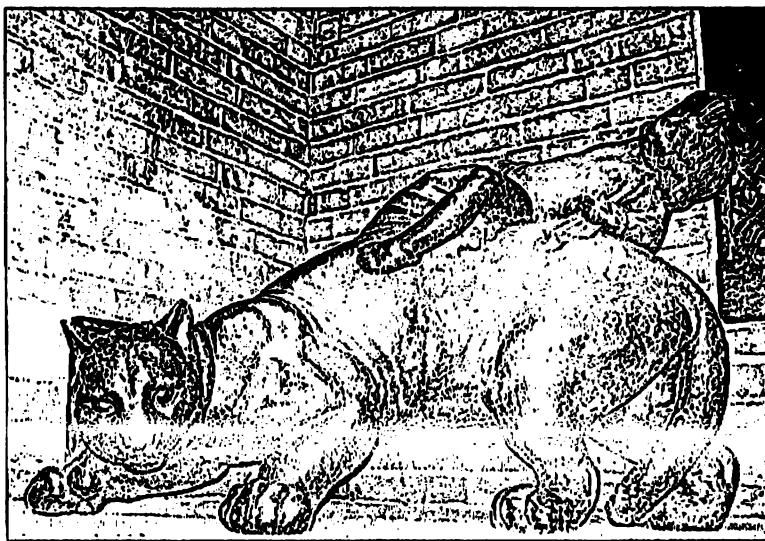
Rezultate PODI

Input Data		PODI				
n	k	RAN	GM	GMS	Δ	LB
1250	315	196813	161427	161481	+44	159040
1250	615	384281	334411	334452	+41	330858
1450	15	54189	2308	2315	+7	1803
1450	315	228251	186626	186702	+76	183975
1450	615	445711	387119	387105	-13	383004
1650	15	12295	2522	2513	-9	1889
1650	315	259779	211935	211933	-2	208841
1650	615	506744	439712	439734	+22	435103
1850	15	13836	2750	2741	-9	1969
1850	315	291321	237099	237102	+3	233542
1850	615	568973	492231	492307	+76	487102
2050	15	15366	2952	2950	-2	1975
2050	315	322770	262273	262282	+9	258412
2050	615	260096	544788	544804	+16	539004

Pentru cazul *PODI*, remarcăm că, în 5 cazuri din 15, *CMS* este mai bun decât *GM* și celelalte rezultate furnizate de *GM* sunt neglijabil mai bune decât cele furnizate de *GMS*. Observăm, de asemenea, că rezultatele furnizate de *RAN* sunt cu atât mai ineficiente, cu cât lungimea k a cuvintelor este mai mică. Ambele *GM* și *GMS* produc rezultate apropiate de algoritmul de mărginire inferioară, *LB*.

Probleme propuse

1. Care sunt relațiile ce trebuie să fie satisfăcute de o metrică? Demonstrați că distanța Hamming este o metrică.
2. Deoarece matricea de adiacență este simetrică și diagonala principală conține numai zerouri, este suficientă doar reținerea elementelor din zona superioară a acesteia. Modificați programul, astfel încât matricea să fie mapată într-un vector care să conțină doar $(n - 1) + (n - 2) + \dots + 1 = n(n - 1)/2$ elemente.
3. Extindeți toate metodele cu validarea parametrilor.
4. Formulați în pseudocod algoritmii *GM*, *GMS* și *LB*.
5. Extindeți programul de mai sus, astfel încât să prezinte pentru fiecare caz de test și raportul detaliat: cuvintele generate, matricea de adiacență, permutările, asignările, arborele parțial minim pentru algoritmii rulați.
6. Extindeți programul, astfel încât să se poată executa și citirea dintr-un fișier a instanțelor problemei.
7. Compactați implementările *GM* și *GMS* prin reținerea completă a informației despre termatorul cuvânt adăugat, fără a mai fi necesar să fie ulterior regăsite.
8. Extindeți programul și cu determinarea timpului de execuție pentru fiecare metodă de rezolvare a *POD/PODI*.
9. Scrieți un algoritm ce determină succesoarea și predecesoarea unei permutări în ordine lexicografică, fără a folosi facilitățile *STL*.
Notă: un astfel de algoritm se găsește de exemplu, în *Log011* în P3.
10. Scrieți programe de test chiar pentru instanțe mai mari ale problemei.
11. Implementați algoritmul *LB* de afilare a arborelui parțial minim, folosindu-l pe cel al lui Prim, în locul celui al lui Kruskal.
12. Citiți în *Help* metodele și exemple, pentru tipul *std::list*.
13. Listați și completați cu noi exemplu toți algoritmii din *library*ul `<algorithm>` utilizati (*random_shuffle*, *find*, *next_permutation*).
14. Metoda *atoi()* transformă un sir de caractere ce reprezintă un număr întreg în numărul întreg corespunzător. Implementați o funcție cu același efect.



Capitolul 8

Recursivitatea

12 probleme complet rezolvate, 36 de probleme propuse

- Inducția matematică
- Recursivitatea. Fundamente
- Suma cifrelor și oglinditul unui număr natural
- Numărul 4
- *Big Mod*
- Tortul (recursivitate liniară)
- Funcția Ackermann (recursivitate îmbricată – compound recursion)
- Transformare recursivă în altă bază (din baza 10 în baza p)
- Sume și produse rădăcini (recursivitate ramificată)
- Funcția lui Collatz (recursivitate nenumărată)
- Pătrale și pătrăjele
- Pătrate (recursivitate directă)
- Pătrate și cercuri (recursivitate indirectă)
- Curba fulgului de zăpadă a lui Koch

Cea ce este slab este ușor de dezbinat. Cea ce este mărunț este ușor de risipit. Trebuie să începeți să facem ordine când încă nu este dezordine. Un copac mare crește din unul mic. Un turn cu nouă etaje începe să se construiască dintr-un pumn de pământ. O călătorie lungă începe cu primul pas.

Lao Zi

Inducția matematică

Deseori se cere demonstrarea unor proprietăți nu numai pentru mulțimi finite de numere, ci pentru mulțimi infinite; de exemplu, pentru mulțimea numerelor naturale, \mathbb{N} . Pentru a demonstra o propoziție pentru toate numerele naturale, nu este posibilă verificarea ei pentru toate elementele din \mathbb{N} , deoarece acest proces nu ar avea sfârșit. Vom descrie în continuare mulțimea numerelor naturale \mathbb{N} folosind axiomele lui Peano, denumite după matematicianul italian Giuseppe Peano (1858-1932), care le-a descoperit.



Giuseppe Peano

- P I: **O este un număr natural.**
- P II: **Fiecare număr natural are exact un succesor, care este tot un număr natural.**
- P III: **O nu este successorul nici unui număr natural.**
- P IV: **Dacă o mulțime X conține numărul 0 și conține succesorul oricărui număr din X, atunci X conține toate numerele naturale.**
- P V: **(axioma inductivă): O mulțime X conține toate numerele naturale atunci când ea conține numărul 0 și pentru fiecare număr din mulțime ea îl conține și pe succesorul acestuia.**

Axioma inductivă este baza pentru metoda inducției complete. De obicei, această metodă este folosită pentru mulțimi infinite (de exemplu, mulțimea numerelor naturale, mulțimea numerelor pare, cea a numerelor prime etc.).

Pașii unei demonstrații prin inducție completă:

- **Baza inducției – BI (pasul 1).** Mai întâi, se demonstrează propoziția pentru cazul de bază (de exemplu, pentru $n = 0$ sau $n = 1$).
- **Premisa inducției – PI (pasul 2).** Presupunem că propoziția $P(n)$ este adevărată pentru un număr n din \mathbb{N} .

- **Concluzia inducției – CI (pasul 3).** Trebuie demonstrat că și $P(n + 1)$ este adevărată, atunci când $P(n)$ este. Cu alte cuvinte, se va demonstra că, dacă propoziția este adevărată pentru un număr natural, atunci ea este adevărată și pentru succesorul acestuia.

Acum se știe că $P(0)$ este adevărată (pasul 1) și concluzia „dacă $P(n)$, atunci și $P(n + 1)$ ” (pasul 3).

Deducem că acest fapt este suficient ca să afirmăm că propoziția este adevărată pentru toate numerele naturale. Ea a fost demonstrată pentru 0 și pe baza concluziei rezultă că ea este adevărată pentru 1, apoi pentru 2 și.a.m.d.

Considerații:

- Concluzia inducției poate să fie adevărată chiar și atunci când baza inducției nu a fost demonstrată. Este necesar însă ca ambele să fie demonstreate.
- Cazul de bază nu trebuie să fie mereu 0. Există propoziții care încep cu un număr mai mare sau mai mic decât 0. Baza inducției trebuie să fie demonstrată pentru cel mai mic număr posibil. Pot să fie chiar mai multe cazuri de bază.
- n este doar o variabilă, care, bineînțeles, poate să se numească și altfel. Pentru propozițiile care conțin mai multe variabile, este necesar să se decidă care dintre ele va fi folosită pentru metoda inducției.

Problema 1. Formula sumei

Demonstrati că $\sum_{k=1}^n \frac{1}{k(k+1)} = \frac{n}{n+1}$ pentru orice număr natural $n \geq 1$.

Soluție: Dacă este cunoscută formula formării sumelor și demonstrarea directă prin dezvoltarea sumei:

$$\sum_{k=1}^n \frac{1}{k(k+1)} = \sum_{k=1}^n \left(\frac{1}{k} - \frac{1}{k+1} \right) = \frac{1}{1} - \frac{1}{2} + \frac{1}{2} - \frac{1}{3} + \dots + \frac{1}{n} - \frac{1}{n+1} = 1 - \frac{1}{n+1} = \frac{n}{n+1}. \square$$

Demonstrație prin inducție completă. Notăm propoziția $\sum_{k=1}^n \frac{1}{k(k+1)} = \frac{n}{n+1}$ cu $P(n)$ și trebuie să demonstrăm că ea este adevărată pentru orice $n \geq 1$.

Baza inducției. $P(1)$ este adevărată: $\frac{1}{1 \cdot 2} = \frac{1}{2}$.

Premisa. $P(n)$: $\sum_{k=1}^n \frac{1}{k(k+1)} = \frac{n}{n+1}$ este adevărată.

Concluzia. Trebuie să demonstrăm că și $P(n+1)$ este adevărată.

Propoziția $P(n+1)$ este: $\sum_{k=1}^{n+1} \frac{1}{k(k+1)} = \frac{n+1}{n+2}$. Calculăm termenul stâng al egalității:

$$\begin{aligned}\sum_{k=1}^{n+1} \frac{1}{k(k+1)} &= \frac{1}{(n+1)(n+2)} + \sum_{k=1}^n \frac{1}{k(k+1)} = \frac{1}{(n+1)(n+2)} + \frac{n}{n+1} = \\ &= \frac{1+2n+n^2}{(n+1)(n+2)} = \frac{(n+1)^2}{(n+1)(n+2)} = \frac{n+1}{n+2}.\end{aligned}$$

Conform principiului inducției matematice, rezultă că propoziția $P(n)$ este adevărată pentru orice număr natural $n \geq 1$. \square

Problema 2. Divizibilitate prin număr prim

Demonstrați că propoziția „Dacă p este un număr prim și n un număr natural, atunci $n^p - n$ se divide cu p ” este adevărată.

Demonstrație prin inducție matematică. Notăm propoziția cu $P(n)$.

Baza inducției. $P(0)$ și $P(1)$ sunt adevărate: $n^p - n = 0$ și 0 divizibil cu p .

Premisa. $P(n)$: „ $n^p - n$ divizibil cu p ” este adevărată.

Concluziu. Trebuie să demonstrăm că și $P(n+1)$ este adevărată.

Cu ajutorul binomului lui Newton, scriem $(n+1)^p - (n+1)$ în următoarea formă:

$$(n+1)^p - (n+1) = (n^p + C_p^1 n^{p-1} + C_p^2 n^{p-2} + \dots + C_p^{p-1} n + 1) - (n+1) =$$

$$= (n^p - n) + C_p^1 n^{p-1} + C_p^2 n^{p-2} + \dots + C_p^{p-1} n. \quad (2)$$

Deoarece p este un număr prim, rezultă că toate $C_p^1, C_p^2, \dots, C_p^{p-1}$ sunt divizibile cu p .

Din premisa inducției, rezultă că și $n^p - n$ este divizibil cu p . Rezultă că toți termenii din partea dreaptă a formulei (2) sunt divizibili cu p , de unde deducem că și $P(n + 1)$ este adevărată.

Conform principiului inducției matematice rezultă că propoziția $P(n)$ este adevărată pentru toate numerele naturale. \square

Problema 3. Inegalitatea Cauchy-Buniakowski-Schwarz

Prezentăm aici o inegalitate utilă, care este utilizată în multe domenii: algebră liniară (vectori), analiză (serii infinite), teoria probabilității, integrarea produselor. Dacă $a_1, a_2, \dots, a_n, b_1, b_2, \dots, b_n$ sunt numere reale, atunci, pentru orice $n \geq 1$, are loc:

$$(a_1^2 + a_2^2 + \dots + a_n^2)(b_1^2 + b_2^2 + \dots + b_n^2) \geq (a_1 b_1 + a_2 b_2 + \dots + a_n b_n)^2. \quad (3)$$

Demonstrație directă. Pentru aceasta, vom folosi următoarea ecuație de gradul al doilea cu necunoscuta x :

$$(a_1 x + b_1)^2 + (a_2 x + b_2)^2 + \dots + (a_n x + b_n)^2 = 0 \quad (4)$$

Ecuția (4) are maxim o rădăcină (partea stângă este mereu mai mare sau egală cu 0 pentru toate numerele reale x) și poate fi scrisă în următoarea formă echivalentă:

$$(a_1^2 + a_2^2 + \dots + a_n^2)x^2 + 2(a_1 b_1 + a_2 b_2 + \dots + a_n b_n)x + (b_1^2 + b_2^2 + \dots + b_n^2) = 0. \quad (5)$$

Pentru că această ecuație are o singură rădăcină, este necesar ca determinantul ei, Δ , să fie mai mic sau egal cu 0:

$$\Delta = 4(a_1 b_1 + a_2 b_2 + \dots + a_n b_n)^2 - 4(a_1^2 + a_2^2 + \dots + a_n^2)(b_1^2 + b_2^2 + \dots + b_n^2) \leq 0. \quad (6)$$

Iar (6) este echivalentă cu (3). \square

Demonstrație prin inducție completă. Notăm (3) cu $P(n)$.

Baza inducției. $P(1)$ este adevărată: $a_1^2 b_1^2 \geq (a_1 b_1)^2 = a_1^2 b_1^2$ pentru orice $a_1, b_1 \in \mathbb{R}$.

Premisa. $P(n)$ este adevărată.

Concluzia. Trebuie să arătăm că și $P(n + 1)$ este adevărată.

$$P(n+1): (a_1^2 + a_2^2 + \dots + a_{n+1}^2)(b_1^2 + b_2^2 + \dots + b_{n+1}^2) \geq (a_1 b_1 + a_2 b_2 + \dots + a_{n+1} b_{n+1})^2$$

$$\Leftrightarrow \begin{cases} (a_1^2 + a_2^2 + \dots + a_n^2)(b_1^2 + b_2^2 + \dots + b_n^2) + a_{n+1}^2(b_1^2 + b_2^2 + \dots + b_n^2) + \\ + b_{n+1}^2(a_1^2 + a_2^2 + \dots + a_n^2) + a_{n+1}^2 b_{n+1}^2 \geq (a_1 b_1 + a_2 b_2 + \dots + a_{n+1} b_{n+1})^2 \end{cases} \quad (7)$$

Pe baza premisei rezultă:

$$\begin{aligned} & \sqrt{a_1^2 + a_2^2 + \dots + a_n^2} \cdot \sqrt{b_1^2 + b_2^2 + \dots + b_n^2} \geq a_1 b_1 + \dots + a_n b_n \\ \Rightarrow & 2a_{n+1} b_{n+1} (a_1 b_1 + \dots + a_n b_n) \leq 2a_{n+1} b_{n+1} \sqrt{a_1^2 + a_2^2 + \dots + a_n^2} \cdot \sqrt{b_1^2 + b_2^2 + \dots + b_n^2} \leq \\ & a_{n+1}^2 (b_1^2 + \dots + b_n^2) + b_{n+1}^2 (a_1^2 + \dots + a_n^2) \end{aligned} \quad (8)$$

Obținem deci:

$$(a_1^2 + a_2^2 + \dots + a_n^2)(b_1^2 + b_2^2 + \dots + b_n^2) \geq (a_1 b_1 + a_2 b_2 + \dots + a_n b_n)^2$$

$$a_{n+1}^2 (b_1^2 + \dots + b_n^2) + b_{n+1}^2 (a_1^2 + \dots + a_n^2) \geq 2a_{n+1} b_{n+1} (a_1 b_1 + \dots + a_n b_n)$$

$$a_{n+1}^2 b_{n+1}^2 = a_{n+1}^2 b_{n+1}^2$$

Ultima inegalitate este adevărată, deoarece

$$2xy \leq x^2 + y^2 \text{ pentru orice } x, y \in \mathbb{R} \\ (\Leftrightarrow (x-y)^2 \geq 0)$$

Atunci când adunăm cele trei relații, obținem:

$$(a_1^2 + a_2^2 + \dots + a_n^2 + a_{n+1}^2)(b_1^2 + b_2^2 + \dots + b_n^2 + b_{n+1}^2) \geq (a_1 b_1 + a_2 b_2 + \dots + a_n b_n + a_{n+1} b_{n+1})^2,$$

și aceasta este $P(n+1)$.

Conform principiului inducției matematice, rezultă că propoziția $P(n)$ este adevărată pentru toate numerele naturale $n \geq 1$. \square

Probleme propuse:

- Demonstrați următoarele propoziții cu ajutorul inducției matematice și scrieți corespunzător programe care le verifică în general:

$$a) \sum_{k=1}^n \frac{k}{(2k-1)(2k+1)(2k+3)} = \frac{n(n+1)}{2(2n+1)(2n+3)} \text{ pentru orice } n \geq 1.$$

$$b) \sum_{k=1}^n \frac{k^4}{(2k-1)(2k+1)} = \frac{n(n+1)(n^2+n+1)}{6(2n+1)} \text{ pentru orice } n \geq 1.$$

- Găsiți formulele generale pentru următoarele formule și demonstrați-le prin inducție:

a) $S_n = 1 \cdot 1! + 2 \cdot 2! + \dots + n \cdot n! = \sum_{k=1}^n k \cdot k!$ pentru orice $n \geq 1$.

b) $P_n = \left(1 - \frac{1}{4}\right) \cdot \dots \cdot \left(1 - \frac{1}{n^2}\right) = \prod_{k=2}^n \left(1 - \frac{1}{k^2}\right)$ pentru orice $n \geq 2$.

3. Fie x un număr real, aşa încât $x \neq \pm 1$. Demonstrați pentru orice $n \in \mathbb{N}$:

$$\frac{1}{1+x} + \frac{2}{1+x^2} + \dots + \frac{2^n}{1+x^{2^n}} = \frac{1}{x-1} + \frac{2^{n+1}}{1-x^{2^{n+1}}}.$$

4. Demonstrați că, pentru orice număr natural $n \geq 1$, are loc:

a) $\cos \alpha \cdot \cos 2\alpha \cdot \dots \cdot \cos 2^n \alpha = \frac{\sin 2^{n+1} \alpha}{2^{n+1} \sin \alpha}$

b) $\sin \alpha + \sin 2\alpha + \dots + \sin n\alpha = \frac{\sin \frac{n+1}{2} \alpha}{\sin \frac{\alpha}{2}} \sin \frac{n\alpha}{2}.$

5. Inegalitatea lui Bernoulli. Demonstrați că, pentru orice număr real x , astfel încât $x > -1$ și $n \in \mathbb{N}$, are loc: $1 + nx \leq (1 + x)^n$.

6. Demonstrați că, pentru orice număr natural $n \geq 1$, au loc:

a) $11^{n+2} + 12^{2n+1}$ divizibil cu 133;

b) $4^n + 15n - 1$ divizibil cu 9;

c) $3 \cdot 5^{2n+1} + 2^{3n+1}$ divizibil cu 17;

d) $7^{n+3} + 3^{2n+1} \cdot 5^{4n+1}$ divizibil cu 23.

7. Jocul lui Hilbert este definit astfel: $F(1) = 1$, $F(2) = 1$, $F(3) = 1$, $F(4) = 1$, $F(5) = 1$, ..., $F(n) = 1$ pentru orice $n \geq 2$. Demonstrați că:

a) $F(n+m) = F(m+1) \cdot F(n) + F(m) \cdot F(n-1)$ pentru orice $m, n \in \mathbb{N} - \{0\}$.

b) $F(n \cdot k)$ este divizibil cu $F(n)$ pentru orice $n, k \in \mathbb{N} - \{0\}$.

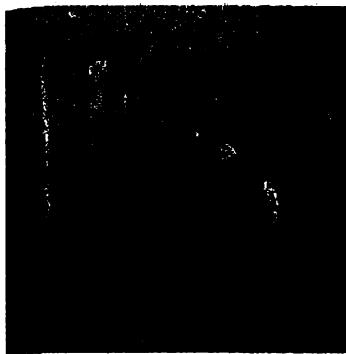
c) $F(kn + k)$ este divizibil cu $F^2(n)$ pentru orice $n \in \mathbb{N} - \{0\}$ și

$$F(kn - 2) \equiv (-1)^{k+1} F^k(n-2) \pmod{F^2(n)}.$$

d) $F_{n \cdot F(n)}$ este divizibil cu $F^2(n)$ pentru orice $n \in \mathbb{N} - \{0\}$.

e) $F(n \cdot F^m(n))$ este divizibil cu $F^{m+1}(n)$ pentru orice $m \in \mathbb{N}$ și $n \in \mathbb{N} - \{0\}$.

Recursivitatea. Fundamente



Termenul recursivitate (în latină, *recurrere* = „a reveni”, „a se întoarce”) înseamnă utilizarea a „ceva” „cu referire la el însuși”. Recursivitatea este pentru informatică ceea ce inducția completă este în matematică. Numim recursivă o structură atunci când părți ale sale au același mod de construcție ca și structura însăși; de exemplu, arborii și listele sunt structuri de date recursive. Un algoritm este recursiv atunci când în interiorul său există subprobleme care apelează la același algoritm. Cunoscuta problemă a turnurilor din Hanoi, de exemplu, se formulează recursiv foarte elegant. Algoritmii iterativi se pot transforma în algoritmi recursivi (pentru o buclă se creează o metodă recursivă) și invers (se folosește o stivă ce reprezintă apelurile recursive).

Pentru a scrie o metodă recursivă, este necesar să decidem ce se întâmplă pentru un anumit nivel, deoarece la toate nivelurile se întâmplă același lucru. Foarte importantă este condiția de terminare (trebuie să fim siguri că algoritmul se termină!); aceasta înseamnă că avem nevoie de o condiție care decide că metoda nu va mai fi

Existe mai multe tipuri de algoritmi recursivi, care pot fi, de exemplu, clasificați după numărul așteptat de apeluri recursive în raport cu dimensiunea variabilelor de control:

- **recursivitate liniară**; fiecare apel al funcției recursive se apelează pe el însuși celuia de control. Exemple: calculul factorialului, suma cifrelor, transformarea unui număr în alt sistem de numerație;
- **recursivitate ramificată**; fiecare apel al funcției recursive conține cel puțin două apeluri ale acelorași funcții. În acest caz, trebuie să simțăm atenții ca rezultatul să nu fie calculată de mai multe ori, situație ce ar conduce la tempi de execuție extrem de mari. Exemple: numerele Fibonacci, coeficienții binomiali;
- **recursivitate imbricată (compound recursion)**; argumentul pentru apelul recursiv apelează însuși din nou metoda recursivă. Astfel, apar foarte multe apeluri ale funcției. Exemple: funcțiile Ackermann, Maniera Ynuclii;

- **recursivitate deschisă (nemonotonă)**: argumentul de control nu se dezvoltă mereu în direcția condiției de terminare. Exemplu: sirul $(3n + 1)$ este cunoscut ca și funcția Collatz.

După tipul apelului, algoritmii recursivi pot fi astfel clasificați:

- **recursivitate directă**: o metodă se apelează direct pe ea însăși.
- **recursivitate indirectă**: cel puțin două metode care se apelează reciproc.

Problema 1. Suma cifrelor și oglinditul unui număr natural

Scrieți metode recursive pentru determinarea sumei cifrelor și a oglinditului unui număr natural dat n ($n > 0$). Oglinditul unui număr natural se obține prin inversarea cifrelor sale. Exemplu:

Tastatură	Ecran
<code>n= 34690213776</code>	<code>sumDigits(n)= 48 reverse(n)= 67731209643</code>

Analiza problemei și proiectarea soluției

Pentru rezolvarea problemei, vom scrie metodele recursive `sumDigits()`, `noDigits()`, `pow()` și `reverse()`. Citiți explicațiile din casetele alăturate. Metoda de oglindire a unui număr natural funcționează astfel:

- Se adaugă ultimei cifre numărul necesar de zerouri.
- Acest număr se adaugă oglinditul numărului obținut prin eliminarea ultimei cifre: $(n \text{ mod } 10) * pow(10, noDigits(n) - 1) + reverse(n / 10)$.

Program

```
#include <iostream>

short sumDigits(unsigned long long n) {
    if(n)
        return (n%10+sumDigits(n/10));
    else
        return 0;
}

short noDigits(unsigned long long n) {
    if(n)
        return 1+noDigits(n/10);
    else
        return 0;
}
```

`short sumDigits(unsigned long long n) {`

`if(n)`

`return (n%10+sumDigits(n/10));`

`else`

`return 0;`

`}`

Suma cifrelor unui număr natural este suma ultimei cifre ($n \text{ mod } 10$) și suma cifrelor numărului obținut prin eliminarea acestei ultime cifre ($n \text{ div } 10$).

`short noDigits(unsigned long long n) {`

`if(n)`

`return 1+noDigits(n/10);`

`else`

`return 0;`

`}`

Numărul cifrelor unui număr natural este 1 plus numărul cifrelor numărului obținut prin eliminarea ultimei cifre ($n \text{ div } 10$).

```

unsigned long long pow(short b, short exp) {
    if(exp)
        return b*pow(b, exp-1);
    else
        return 1;
}

unsigned long long reverse(unsigned long long n) {
    if(n)
        return ((n%10)*pow(10,noDigits(n)-1)+reverse(n/10));
    else
        return 0;
}

int main(){
    unsigned long long n;
    std::cout << "n= " ; std::cin >> n;
    std::cout << "sumDigits(n)= " << sumDigits(n) << std::endl;
    std::cout << "reverse(n)= " << reverse(n);
    return 0;
}

```

$$b^n = \begin{cases} 1, & \text{pentru } n=0 \\ b \cdot b^{n-1}, & \text{pentru } n>0 \end{cases}$$

Probleme propuse

1. Scrieți pe hârtie valorile pentru pașii execuției de program relativ la exemplul dat.
2. Implementați metode iterative pentru metodele recursive *sumDigits()*, *noDigits()*, *pow()* și *reverse()*.
3. Implementați metoda de ridicare rapidă la putere, pe baza algoritmului următor:

Ridicarea rapidă la putere ($O(\log n)$, unde n este exponentul)

$$1. e = e_0 + e_1 \cdot 2 + e_2 \cdot 2^2 + \dots + e_r \cdot 2^r$$

... unde $e_i \in \{0, 1\}$ și este numărul binar al lui e

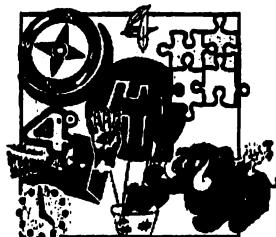
$$2. e = e_0 + 2 \cdot (e_1 + 2 \cdot (e_2 + 2 \cdot (\dots + 2 \cdot (e_{r-1} + 2 \cdot e_r)) \dots))$$

$$3. b^e = (((((b^{e_r})^2 \cdot b^{e_{r-1}})^2 \cdot b^{e_{r-2}})^2 \cdot b^{e_{r-3}}) \dots)^2 \cdot b^{e_0}$$

4. La fiecare pas, rezultatul parțial este ridicat la putere și apoi multiplicat cu 1 sau cu b , după cum e_i este 0, respectiv 1.

4. Scrieți metode iterative și recursive pentru calculul valorilor $n! = 1 \cdot 2 \cdots n$ și $S(n) = 1 + 2 + \dots + n$.
5. Scrieți o metodă iterativă și una recursivă pentru calculul sumei $S(m, n) = m + (m+1) + \dots + n$ (dacă $m > n$, atunci returnați 0).

Problema 2. Numărul 4



Numărul patru simbolizează cele patru puncte cardinale, cele patru anotimpuri și cele patru elemente.

Se poate demonstra că din numărul patru se poate obține orice număr natural cu ajutorul următoarelor operații:

- A) se adaugă un 4 la sfârșit;
- B) se adaugă un 0 la sfârșit;
- C) numărul curent se împarte la 2.

De exemplu, numărul 2354 se obține din numărul 4 aplicând succesiv operațiile CCBCBACA. Scrieți un program care să arate pentru numerele naturale din fișierul nr4.in cum se pot obține acestea succesiv din numărul 4, folosind operațiile A-C. Rezultatele se vor scrie în fișierul nr4.out. Exemplu:

nr4.in	nr4.out
2524	4->2->1->10->5->50->504->252->2524
564	4->44->22->224->112->56->564
12	4->2->24->12
3	4->2->24->12->6->3

(Inspirată din revista sovietică Kvant, autor A.K. Tolpigo)

Analiza problemei și proiectarea soluției

Vom determina drumul către numărul n prin construirea drumului invers de la n la numărul 4, ceea ce implica înversarea lui.

- A') se elimină cifra 4 de la sfârșit, dacă este un 4;
- B') se elimină cifra 0 de la sfârșit, dacă este 0;
- C') se înmulțește cu 2.

Denumirea este să se obțină:

operațiilor A-C.

Puteam să clasificăm numărul n după ultima cifră și să demonstrăm pentru fiecare categorie. Este însă suficient dacă demonstrăm că din orice număr par $n \geq 4$ se poate ajunge la un număr par mai mic decât el. Dacă n este un număr impar, atunci el se poate transforma în număr par prin înmulțirea cu 2. Dupa ultima cifră a numărului par n , sunt posibile următoarele transformări:

- i) $10k \rightarrow k$;
- ii) $10k + 2 \rightarrow 20k + 4 \rightarrow 2k$;
- iii) $10k + 4 \rightarrow k \rightarrow 2k$;
- iv) $10k + 6 \rightarrow 20k + 10 + 2 \rightarrow 40k + 20 + 4 \rightarrow 4k + 2$;
- v) $10k + 8 \rightarrow 20k + 10 + 6 \rightarrow 40k + 30 + 2 \rightarrow 80k + 60 + 4 \rightarrow 8k + 6$. □

Vom implementa metoda recursivă `numberFour()`.

Program

```
#include <iostream>

void numberFour(int n, std::ofstream &out){
    if(n==4){
        switch(n%10){
            case 0: numberFour(n/10, out); break;
            case 4: numberFour(n/10, out); break;
            default: numberFour(n*2, out);
        }
        out << " ->" << n;
    }
}

int main(){
    int n;
    std::ifstream in("nr4.in");
    std::ofstream out("nr4.out");
    while(in && !in.eof() && in>>n){
        out << "\n4";
        numberFour(n, out);
    }
    return 0;
}
```

Probleme propuse

- Notați pe hârtie pașii făcuți de algoritm pentru numerele din exemplu.
- Scrieți o metodă iterativă pentru rezolvarea problemei.

Problema 7. Big Mod

Calculați valoarea $R = B^P \bmod M$ pentru B , P și M numere naturale, așa încât $0 \leq B, P \leq 200.000.000$ și $0 \leq M \leq 50.000$.

Date de intrare. În fișierul `bigmod.in` se găsesc mai multe instanțe ale problemei ca triplete, cîte unul pe linie.

Date de ieșire. Scrieți în fișierul de ieșire `bigmod.out` pentru fiecare triplu (B, P, M) valoarea $R = B^P \bmod M$. Exemplu:

<code>bigmod.in</code>	<code>bigmod.out</code>
3 18132 17	13
17 1765 3	4
2374859 3029382 36123	13195

Analiza problemei și proiectarea soluției

Propoziție. Pentru orice numere $a, b, c \in \mathbb{N} - \{0\}$, are loc $(a \cdot b) \bmod c = ((a \bmod c) \cdot (b \bmod c)) \bmod c$.

Demonstrație. Conform teoremei împărțirii cu rest, obținem succesiv:

$$\begin{aligned} \exists R \in \{0, \dots, c-1\}, \text{a.f. } a \cdot b = Q \cdot c + R \rightarrow \\ R = (a \cdot b) \% c = a \cdot b - Q \cdot c \end{aligned} \quad (1)$$

$$\begin{aligned} \exists R_1 \in \{0, \dots, c-1\}, \text{a.f. } a = Q_1 \cdot c + R_1, R_1 = a \% c \\ \exists R_2 \in \{0, \dots, c-1\}, \text{a.f. } b = Q_2 \cdot c + R_2, R_2 = b \% c \Rightarrow a \cdot b = Q_1 \cdot c + R_1 \cdot R_2 \Rightarrow \end{aligned}$$

$$\Rightarrow (a \cdot b) \% c = (R_1 \cdot R_2) \% c = (a \% c \cdot b \% c) \% c. \square$$

Vom scrie metoda recursivă `bigMod()`, care se bazează pe această propoziție:

$$\text{bigMod}(B, P, M) = \begin{cases} \text{a)} & 1, \text{ dacă } P = 0 \\ \text{b)} & 0, \text{ dacă } a \text{ neadevărată și } B = 0 \\ \text{c)} & B \% M, \text{ pentru } a \text{ și } b \text{ neadevărate și } (B = 0 \text{ sau } P = 1) \\ \text{d)} & (\text{bigMod}(B, P/2, M)^2 \% M, \text{ pentru } a, b \text{ și } c \text{ neadevărate și } P \text{ număr par} \\ \text{e)} & (\text{bigMod}(B, P - 1, M) \cdot (B \% M)) \% M, \text{ pentru } a, b, c \text{ neadevărate și } P \text{ număr impar} \end{cases}$$

Program

```
#include <iostream>

using namespace std;

unsigned long bigMod(unsigned long B, unsigned long P, unsigned long M) {
    if(0==P) return 1;
    if(0==B) return 0;
    if(1==B || 1==P) return B%M;
    if(1==M) return 0;
    if(0 == P%2) {
        unsigned long aux;
        aux = bigMod(B, P/2, M);
        return (aux*aux)%M;
    } else
        return (bigMod(B, P-1, M) * (B%M))%M;
}
```

```

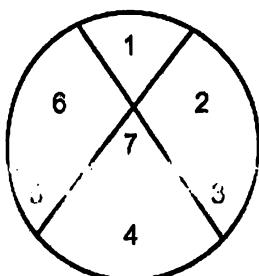
int main(){
    unsigned long int B, P, M;
    ifstream in("bigmod.in");
    ofstream out("bigmod.out");
    while(in && !in.eof() && in>>B>>P>>M)
        out << bigMod(B, P, M) << endl;
    return 0;
}

```

Probleme propuse

- Scriți și o variantă iterativă pentru rezolvarea problemei.
- Demonstrați propoziția de mai sus astfel: $n \bmod c = n - c(n \text{ div } c)$, $((a \bmod c) \times (b \bmod c)) \bmod c = ((a - c(a \text{ div } c)) \times (b - c(b \text{ div } c))) = \dots$
- De la tastatură se citesc n numere naturale a_1, a_2, \dots, a_n , fiecare mai mic decât 2000000, și un număr natural $0 < c < 50000$. Scriți un program care calculează valoarea $(a_1 \cdot a_2 \cdot \dots \cdot a_n) \bmod c$.

Problema 4. Tortul (recursivitate liniară)



La o petrecere, trebuie tăiate bucăți de tort care pot avea dimensiuni diferite. Din trei tăieturi, se pot obține șapte bucăți, ca în figura alăturată. Câte bucăți se pot obține din n tăieturi? (Câte regiuni se obțin maximal în plan atunci când se trasează n drepte?)

Date de intrare. În fișierul `tort.in`, pentru fiecare linie, numarul de tăieturi n ($0 \leq n \leq 2000$).

Date de ieșire. Scriți în fișierul `tort.out`, pentru fiecare număr de tăieturi din fișierul de intrare, numărul maxim de bucăți de tort care se pot obține cu

<code>tort.in</code>	<code>tort.out</code>
0	0 tăieturi -> 1 bucăți!
1	1 tăieturi -> 2 bucăți!
2	2 tăieturi -> 4 bucăți!
9	9 tăieturi -> 46 bucăți!
10	10 tăieturi -> 56 bucăți!
2000	2000 tăieturi -> 2001001 bucăți!
678	678 tăieturi -> 230182 bucăți!

Analiza problemei și proiectarea soluției

Dacă numărul maxim de regiuni pentru $n - 1$ drepte care se intersectează este $S(n - 1)$, atunci pentru încă o dreaptă se vor obține în plus încă n regiuni:

$S(0) = 1$ (nici o tăietură \rightarrow rămâne o bucată; întregul tort)

$S(n) = n + S(n - 1)$, pentru orice $n > 0$. (1)

Formula (1) este o formulă recursivă și vom implementa corespunzător ei metoda $S()$ în program.

Program 4.1.

```
#include <iostream>

int S(int n){
    if(n<1) return 1;
    else
        return n + S(n-1);
}

int main(){
    int n;
    std::ifstream in("tort.in");
    std::ofstream out("tort.out");
    while(in && !in.eof() && in>>n){
        out.width(1);
        out << n << " tăieturi -> ";
        out.width(7);
        out << S(n) << " bucati!" << std::endl;
    }
    return 0;
}
```

Observăm că, pentru valori mai mari ale lui n , timpul de execuție al programului crește foarte mult, deoarece metoda se aplicează foarte des. Din formula (1) deducem:

$$S(n) = n + S(n - 1) = n + (n - 1) + S(n - 2) = \dots = 1 + \frac{n \cdot (n+1)}{2}. \quad (1)$$

Program 4.2.

```
#include <iostream>

int main(){
    long n;
    std::ifstream in("tort.in");
    std::ofstream out("tort.out");
    while(in && !in.eof() && in>>n && n >=0){
        out.width(8);
        out << n << " tăieturi -> ";
        out.width(10);
        out << 1 + n*(n+1)/2;
        << " bucati!" << std::endl;
    }
    return 0;
}
```

Problemă propusă

Problema inversă. Se dă numărul maxim de regiuni care se obțin și se cere numărul de tăieturi. Toate numerele în fișierul *reg.in* sunt, aşadar, de forma $1 + \frac{n(n+1)}{2}$. Exemplu:

reg.in	tăieturi.out
56	10
2001601	1000
2301#2	?

Problema 5. Funcția Ackermann

(recursivitate imbiricată – compound recursion)

Există mai multe funcții matematice care poartă denumirea de Ackermann. Inițial, o astfel de funcție ce crește extrem de repede a fost introdusă de către Wilhelm Ackermann în 1926. Funcția Ackermann este folosită de exemplu pentru determinarea limitelor în informatică teoretică. Cu ajutorul ei, se pot construi module de test pentru apeluri recursive.

Funcția lui Ackermann este astfel definită pentru $m, n \in \mathbb{N}$:

$$Ack(m, n) = \begin{cases} m + 1, & \text{dacă } n = 0 \\ Ack(m - 1, 1), & \text{dacă } n = 1 \\ Ack(m - 1, Ack(m, n - 1)), & \text{dacă } m \neq 0 \text{ și } n \neq 0 \end{cases}$$

Funcția lui Ackermann crește foarte repede: $Ack(3, 4) = 125$, dar $Ack(4, 2)$ are 19729 cifre! Scrieți un program care să calculeze valoarea funcției lui Ackermann pentru perechi (n, m) de numere naturale, știind că această valoare se încadrează în tipul *unsigned long long*. Exemplu:

ack.in	ack.out
0 2	Ack(0, 2)= 3
2 0	Ack(2, 0)= 3
3 4	Ack(3, 4)= 125
3 5	Ack(3, 5)= 253
3 2	Ack(3, 2)= 29

Analiza problemei și proiectarea soluției

Vom implementa definiția matematică recursivă a funcției cu ajutorul metodei `ack()`.

Program

```

#include <fstream>

unsigned long long ack(short n, short m){
    if(0==n) return m+1;
    else if(0==m) return ack(n-1, 1);
    else return ack(n-1, ack(n, m-1));
}

int main() {
    short m, n;
    std::ifstream in("ack.in");
    std::ofstream out("ack.out");
    while(in && !in.eof() && in>>n>>m) {
        out << "Ack(" << n << ", " << m
    }
    return 0;
}

```

Problema នរបាល

Manna-Pnueli. Scrieți un program care pentru un număr întreg x , astfel încât $-1000 \leq x \leq 1000$, calculează valoarea funcției Manna-Pnueli $f(x)$, definită astfel:

$$f: \mathbb{Z} \rightarrow \mathbb{Z}, f(x) = \begin{cases} x-1, & \text{dacă } x > 1 \\ f(f(x+2)), & \text{dacă } x < 12 \end{cases}$$

Exemplu:

mp.in	mp.out
12	Manna-Pnueli(12) = 11
34	Manna-Pnueli(34) = 33
56	Manna-Pnueli(56) = 55
-123	Manna-Pnueli(-123) = 11
98	Manna-Pnueli(98) = 97
678	Manna-Pnueli(678) = 677
-234	Manna-Pnueli(-234) = 11

Problema 6. Transformare recursivă în altă bază (din baza 10 în baza p)

Scrieti o metodă recursivă care să transforme un număr natural n ($0 \leq n \leq 9.000.000$) din baza 10 într-o bază dată p ($q \leq p \leq 9$) Exemplu:

baseP.in	baseP.out
324123534 2	324123534 in base 2 = 10011010100011011101110001110
324539 9	324539 in base 9 = 544158
654789045 6	654789045 in base 6 = 1445502222433
675432 4	675432 in base 4 = 2210321220
9000000 8	9000000 in base 8 = 42252100
3 2	3 in base 2 = 11
8 6	8 in base 6 = 12

Analyze problema și propoziția soluției

Dacă n are reprezentarea $a_k a_{k-1} \dots a_1 a_0$ în baza p , atunci are loc:

$$n = \overline{a_k a_{k-1} \dots a_1 a_0} = a_k p^k + a_{k-1} p^{k-1} + \dots + a_1 p + a_0, \quad a_k, a_{k-1}, \dots, a_0 \in \{0, 1, \dots, p-1\}$$

$$\text{Deci } n = p(a_k p^{k-1} + a_{k-1} p^{k-2} + \dots + a_1) + a_0.$$

De aici, se deduce următorul algoritm recursiv:

```

doTransformBase10ToP(n, p)
  If (p > 0) Then
    doTransformBase10ToP(n div p, p)
    Write (n % p)
  End_If
End_doTransformBase10ToP(n, p)

```

Program

```
#include <iostream>
using namespace std;

void transfBase10ToP(unsigned long long n,
                     short p, ofstream &out){
    if(n){
        transfBase10ToP(n/p, p, out);
        out << n%p;
    }
}

int main(){
    unsigned long long n;
    short p;
    ifstream in("baseP.in");
    ofstream out("baseP.out");
    while(in && !in.eof() && in>>n>>p){
        out.width(10);
        out << n << " in base " << p
            << " = ";
        transfBase10ToP(n, p, out);
        out << endl;
    }
    return 0;
}
```

Probleme propuse

- Scriți un algoritm nerecursiv care să rezolve problema.
- Implementați un program (variantă recursivă și iterativă) pentru transformarea inversă. Se dă o pereche de numere (n_p, p) și se cere transformarea din baza p în baza 10. Exemplu:

baseP10.in	baseP10.out
42282170 11 2 12 6	3000000 3 8

Analizați următoarea implementare:

```
unsigned long long getBasePTo10(string np, short p) {
    if(np.length()){
        short aux = np[np.length()-1]-'0';
    }
```

```

    return aux +
    p*getBasePTo10(np.substr(0, np.length()-1), p);
} else
    return 0;
}

```

Valorile ASCII '0', '1', ..., '9' sunt amplasate succesiv și de aceea valoarea $(np[np.length() - 1] - '0')$ este de fapt numărul corespunzător 0, 1, 2, ..., 9.

3. Scrieți cel puțin două variante pentru transformarea unui număr natural în binar prin folosirea operatorilor pe biți. Analizați varianta:

```

cin >> n; cout << n << " in base 2 = ";
short SIZ_INT = sizeof(unsigned long long)*8;
for(short i=SIZ_INT-1; i>=0; i--)
    cout << (n>>i&1);                                //bitul al i-lea din n

```

Problema 7 Suma a două rădăcini (recursivitate ramificată)

Se consideră ecuația $x^2 - Sx + P = 0$, cu $S, P \in \mathbb{R}$. Fie x_1 și x_2 rădăcinile acestei ecuații și trebuie calculată valoarea $T_n = x_1^n + x_2^n$ pentru $n \in \mathbb{N}$. T_n va trebui calculat fără a folosi valorile explicite ale rădăcinilor x_1 și x_2 . Numerele reale S, P , cu $-500 \leq S, P \leq 500$, și numărul natural n ($0 \leq n \leq 20$) se află în fișierul de intrare, iar T_n trebuie scris în fișierul de ieșire, câte unul pe linie pentru fiecare set de date de intrare. Exemplu:

Input	Output
5 6.54 3.22	7799.79
0 3 4	2
1 3.45 6.78	3.45

Analiza problemei și proiectarea soluției

Deoarece x_1 și x_2 sunt rădăcinile ecuației, rezultă că $x^2 - Sx + P = (x - x_1)(x - x_2)$.

Deci $x_1 + x_2 = S$ și $x_1 \cdot x_2 = P$ (relațiile lui Viète). Așadar, formula pentru T_n este:

$$T_n = \begin{cases} 2, & \text{dacă } n = 0 \\ x_1 + x_2, & \text{dacă } n = 1 \\ ST_{n-1} - PT_{n-2}, & \text{dacă } n > 1 \end{cases} \quad (1)$$

Program

```
#include <fstream>

double sum(short n, double S, double P){
    if(0==n) return 2;
    if(1==n) return S;
    return
        S*sum(n-1, S, P)-P*sum(n-2, S, P);
}

int main(){
    short n;
    double S, P;
    std::ifstream in("equation.in");
    std::ofstream out("equation.out");
    while(in && !in.eof() && in>>n>>S>>P){
        out.precision(6);
        out << sum(n, S, P) << std::endl;
    }
    return 0;
}
```

Probleme propuse

1. Demonstrați formula (1) prin inducție.
 2. Generalizare. Se consideră ecuația de gradul k : $x^k - S_1x^{k-1} + \dots + (-1)^k S_k = 0$ cu rădăcinile x_1, x_2, \dots, x_k . Arătați că valoarea $T_n = x_1^n + x_2^n + \dots + x_k^n$ este constantă.
- Indicație.* Formula recursivă este $T_n = S_1T_{n-1} - S_2T_{n-2} + \dots + (-1)^{k-1} \cdot S_k \cdot T_{n-k}$.

Problema 8. Funcția lui Collatz (recursivitate nemonotonă)

Într-o secvență de numere naturale se apără o proprietate interesantă:

$$f(n) = \begin{cases} 1, & \text{pentru } n = 1 \\ \frac{n}{2}, & \text{dacă } n \text{ este par} \\ 3 \cdot n + 1, & \text{dacă } n \text{ este impar} \end{cases}$$

și are proprietatea că este „convergentă” către 1. De exemplu, pentru $n = 12$, secvența generată este: 12, 6, 3, 10, 5, 16, 8, 4, 2, 1 și are lungimea 9, deoarece se ajunge în 9 pași la valoarea 1. Scrieți o funcție recursivă, care generează secvența corespunzătoare și afișează și numărul de pași în care s-a ajuns la 1. Exemplu:

numbers.in	collatzSeq.out
1	1 [1] STOP<0>
12	12 6 3 10 5 16 8 4 2 1 [1] STOP<9>
67	67 202 101 304 152 76 38 19 58 29 88 44 22 11 34 17 52
1003	26 13 40 20 10 5 16 8 4 2 1 [1] STOP<27>
234	1003 3010 1505 4516 2258 1129 3388 1694 847 2542 1271 3814 1907 5722 2861 8584 4292 2146 1073 3220 1610 805 2416 1208 604 302 151 454 227 692 341 1024 512 256 128 64 32 16 8 4 2 1 [1] STOP<41>
	234 117 352 176 88 44 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1 [1] STOP<21>

Analiza problemei și proiectarea soluției

Vom implementa metoda recursivă *fCollatz()*, care, cu ajutorul parametrului *l*, numără și numărul de apeluri ale funcției.

Program

```
#include <iostream>

using namespace std;

void fCollatz(unsigned long n, int& l, ofstream &out){
    out << n << " ";
    if(n!=1){
        l++;
        if(n%2){
            fCollatz(3*n+1, l, out);
        }else{
            l--;
        }
    } else{
        out << "[1] STOP";
    }
}

int main(){
    unsigned long n;
    int l;
    ifstream in("numbers.in");
    ofstream out("collatzSeq.out");
    while(in && !in.eof() && in>>n){
        l=0;
        fCollatz(n, l, out);
    }
}
```

```

    fCollatz(n, l, out);
    out << "<" << l << ">" << endl << endl;
}
return 0;
}

```

Probleme propuse

1. Scrieți și o variantă recursivă.
2. Notăm cu A operația de „înjunătățire” și cu B operația „ $3n + 1$ ”. Dezvoltați programul astfel încât să calculeze și numărul de operații de fiecare tip. De exemplu, pentru $n = 12$ este nevoie de șapte operații A și de două operații B ; pentru $n = 1003$ este nevoie de 29 operații A și de 12 operații B . Implementați un program care să prelucreze toate numerele dintr-un interval dat și să le afișeze pe cele care au nevoie de un număr maximal de pași pentru a ajunge la 1 (de exemplu, în intervalul $[45, 459]$ se găsește numărul 327, care are nevoie de 143 de pași). Furnizați de asemenea acele numere din intervalul dat care au nevoie de un număr maxim de operații A , respectiv B .

Problema 9. Pătrate și pătrătele

Un pătrat cu laturile paralele cu axele este unic, dacă se dă centrul său și lungimea laturii. Spunem că un pătrat cu lungimea laturii $2k+1$ are dimensiunea k . Pentru un număr natural k , definim familia păratelor cu următoarele proprietăți:

- i) cel mai mare pătrat are dimensiunea k (lungimea laturii sale este $2k+1$) și se află amplasat în centrul tuturor rețelei de dimensiune 1024 (aceasta înseamnă lungimea laturii 2049);
- ii) pentru dimensiunea k , are loc: $1 \leq k \leq 512$;
- iii) toate pătrătele care au o dimensiune mai mare decât 1 genereză câte un pătrat de dimensiune $k \text{ div } 2$ în toate cele patru vârfuri ale sale;
- iv) colțul stânga-sus al ecranului are coordonatele $(0, 0)$ și colțul dreapta-jos are coordonatele $(2048, 2048)$.

Pentru un număr natural k , putem să definim în mod unic familia păratelor cu aceste proprietăți. Un punct pe ecran se poate găsi în nici unul, unul sau mai multe pătrăte. Scrieți un program care, pentru un număr natural k dat și coordonatele unui punct în plan, să decidă în câte pătrăte din familia k se află acest punct.

Date de intrare. Fiecare linie în fișierul de intrare *patrate.in* conține trei valori k , x_0 și y_0 și reprezintă un set de date de intrare.

Date de ieșire. Scrieți pentru fiecare caz din fișierul de intrare câte o linie în fișierul de ieșire *patrate.out* cu numărul corespunzător de pătrăte în care se găsește punctul (x_0, y_0) . Exemplu:

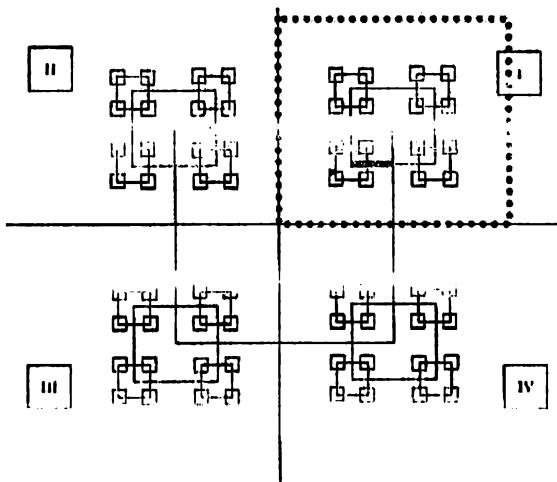
patrate.in	patrate.out
56 1012 1000	1
500 1000 1000	4
500 113 941	5
1100 512 512	5

(ACM, North-Western European Regionals, 1992)

Analiza problemei și proiectarea soluției

O primă metodă de rezolvare ar fi să verificăm recursiv pentru fiecare pătrat dacă el conține punctul dat, caz în care variabila n se va incrementa. Pentru aceasta, vom scrie metoda $\text{inSquare}(x_0, y_0, cx, cy, k)$, unde primii doi parametri definesc punctul (x_0, y_0) și următorii definesc pătratul cu centru (cx, cy) și dimensiunea k . Moștă $\text{count}()$ numără în câte pătrate se află punctul dat și este apelată recursiv pentru cele patru pătrate care se creează în cele patru vârfuri ale păratului dat.

Propoziție. Dacă desenăm și axe de simetrie prin centrul primului pătrat, atunci toate pătratele generate de vârful dreapta-sus se află în primul cadran. La fel, toate pătratele generate de vârful stânga-sus se află în cel de-al doilea cadran, cele generate de vârful stânga-jos în al treilea, cele generate de vârful dreapta-jos se află în cel de-al patrulea cadran.



Demonstrație. În cazul în care k este lungimea laturii păratului punctat, atunci segmentele îngroșate care se apropie de ordonată au lungimile $\frac{k}{2}, \frac{k}{4}, \frac{k}{8}, \dots, \frac{k}{2^n}, \dots$

Acestea reprezintă o progresie geometrică; rezultă deci:

$$\frac{k}{2} + \frac{k}{2^2} + \frac{k}{2^3} + \dots + \frac{k}{2^n} = k \cdot \left(1 - \frac{1}{2^n}\right).$$

Sirul $k \cdot \left(1 - \frac{1}{2^n}\right)$, pentru $n = 0, 1, 2, \dots$, este crescător și are limita k , de unde rezultă că toate pătratele generate de vârful dreapta-sus se află în primul cadran. Datorită simetriei, deducem de asemenea că și toate celelalte grupe de pătrate se află în cadranele II, III și IV. \square

A două alternativă pentru program apelează metoda recursivă doar pentru pătratul în care se află punctul dat.

Program

```
#include <fstream>

int inSquare(int x0, int y0, int cx, int cy, int k) {
    int x1 = cx - k;
    int x2 = cx + k;
    int y1 = cy - k;
    int y2 = cy + k;
    return (x1 <= x0 && x0 <= x2 &&
            y1 <= y0 && y0 <= y2);
}

void count(int x0, int y0, int cx, int cy, int k, int& n) {
    int x1, x2, y1, y2;
    if(inSquare(x0, y0, cx, cy, k)) n++;
    if(k>1) {
        x1 = cx - k;
        y1 = cy - k;
        x2 = cx + k;
        y2 = cy + k;
        count(x0, y0, x1, y1, k/2, n); ...
        if(inSquare(x0, y0, x1, y1, k))
            count(x0, y0, x1, y1, k/2, n);
        if(inSquare(x0, y0, x1, y2, k))
            count(x0, y0, x1, y2, k/2, n);
        if(inSquare(x0, y0, x2, y1, k))
            count(x0, y0, x2, y1, k/2, n);
        if(inSquare(x0, y0, x2, y2, k))
            count(x0, y0, x2, y2, k/2, n);
        ...
    }
}
```

```

int main(){
    int k, x0, y0, n;
    std::ifstream in("quadrate.in");
    std::ofstream out("quadrate.out");
    while(in && !in.eof() && in>>k>>x0>>y0){
        n=0;
        count(x0, y0, 1024, 1024, k, n);
        out<<n<<std::endl;
    }
    return 0;
}

```

Probleme propuse

1. Scrieți și o variantă iterativă ce rezolvă problema dată.
2. Calculați suma ariilor tuturor pătratelor în care se află punctul dat.

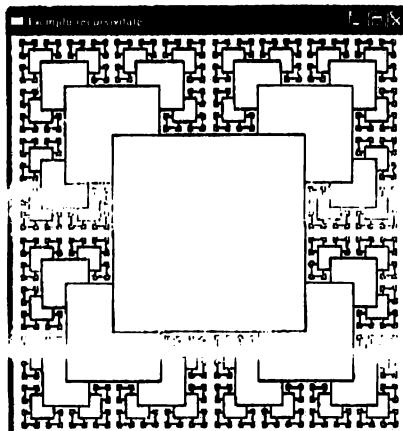
Problema 10. Pătrate (recursivitate directă)

Scrieți o aplicație recursivă care desenează figura alăturată.

Analiza problemei și proiectarea soluției

În vârfurile unui pătrat cu lungimea laturii L se desenează câte un pătrat cu lungimea laturii $L/2$. Aceste patru pătrate generate au centrele în vârfurile păratului initial. Pentru toate cele patru pătrate astfel desenate, se vor genera mai departe 16 pătrate cu lungimea laturii $L/4$ și centrele în vârfurile lor. Procedeul se repetă până când lungimea

deținute de patru pătrate este mai mică decât 10 pixeli. Pentru obținerea figurii de mai sus, avem nevoie de o bibliotecă grafică C++, care, de obicei, este implementată diferit pe sisteme de operare diferite. Vom prezenta două variante de implementare. Prima este simplă și ușor de înțeles, folosind compilatorul Borland C++ 3.1.



Program 1 (Borland C++ 3.1)

```

#include <graphics.h>
#include <iostream.h>
#include <conio.h>
#include <stdlib.h>

```

```

void initGrafix()
{
    int gdet = DETECT, gm, err;
    initgraph(&gdet, &gm, "c:\\borlandc\\BGI");
    err = graphresult();
    if(err != grOk){
        cout<< "Grafic error: "<< grapherrmsg(err)<< endl;
        cout<< "Press any key to quit.";
        getch(); exit(1);
    }
}

void drawSquare(int x, int y, int l)
{
    if(l>50){
        setcolor(14);
        drawSquare(x-1/2, y-1/2, l/2);
        drawSquare(x-1/2, y+1/2, l/2);
        drawSquare(x+1/2, y-1/2, l/2);
        drawSquare(x+1/2, y+1/2, l/2);
        rectangle(x-1/2, y-1/2, x+1/2, y+1/2);
    }
}

int main(){
    initGrafix();
    drawSquare (getmaxx()/2, getmaxy()/2, getmaxy()/2);
    ...();
    closegraph();
    return 0;
}

```

În cadrul acestui proiect se va folosi o interfață Windows, care folosește biblioteca *Active Template Library (ATL)*. ATL este o bibliotecă de clase bazată pe şablonare (*template*), care ajută la crearea și manipularea ferestrelor, a dialogurilor, a elementelor de control etc. Folosirea bibliotecii ATL ușurează programarea directă Windows, care este mai grea și folosește concepție avansată, cum ar fi moștenire.

Folosirea ferestrelor în ATL se bazează pe clasa *CWindow*, care conține un Windows-handle (*HWND*) și pe aproape toate metodele *User32* care așteaptă ca prim parametru *HWND*. Metoda specifică Windows, *BOOL ShowWindow (HWND hWnd, int nCmdShow)*, are în ATL forma: *BOOL CWindow::ShowWindow (int nCmdShow)*.

- *CWindowImpl* – înregistrează clasa corespunzătoare ferestrei și *message handling*;
- *CDialogImpl* – oferă posibilitatea de a crea ferestre de dialog modale și nemodale;

- *CSimpleDialog* și *CDialogImpl* – specializări ale dialogurilor modale, ce conțin funcționalități de bază.

În afara claselor corespunzătoare ferestrelor, există și clase ajutătoare, ce ușurează implementarea obiectelor ATL de tip fereastră:

- *CWndClassInfo* – oferă metode de înregistrare pentru informațiile unei ferestre;
- *CWinTraits* (sau *CWinTraitsOR*) – oferă metode simple pentru standardizarea caracteristicilor unui obiect de tip fereastră din ATL.

Pentru a putea folosi ATL, trebuie incluse antetele *atlwin.h* și *atldbase.h*. Acestea ar trebui să fie incluse în header-ul (antetul) precompilat.

Pentru versiunile ATL mai vechi de 7.0 este necesară declararea unei instanțe *CComModule*. Ea acționează ca o instanță globală, al cărei scop principal este acceptarea unei instanțe *HINSTANCE* pentru aplicație. Aplicația trebuie să apeleze la *Inceput_Module.Init(...)* pentru a crea o instanță *HINSTANCE* pentru *DLLMain* sau *WinMain*. La sfârșit, trebuie apelată *_Module.Term()* pentru a elibera seturile de date.

Folosirea lui *CComModule* pentru ATL 7.0 este depășită (obsolete).

Dacă programul se compilează cu versiuni ATL mai vechi (de exemplu, ATL 3.0 din Visual C++ 6.0), se poate utiliza macroul *_ATL_VER* în header-ul precompilat *stdafx.h* cu forma:

```
#include <atldbase.h>

#if _ATL_VER < 0x0700
extern CComModule _Module;
#endif

#include <atlwin.h>

```

În acest fel, trebuie procedat cu *_Module.Init(...)* și *_Module.Term(...)* din *WinMain*.

Prezintă în continuare o altă soluție pentru problema dată, folosind ATL: antetul precompilat *stdafx.h*.

```
#ifndef _STDAFX_H_
#define _STDAFX_H_ ...

#include <atldbase.h>
#include <atlwin.h>
#endif
```

Stabilește folosirea ATL.

Antetul *MainWindow.h* definește în clasa abstractă *CMainWindow* fereastra principală a aplicației și declară o funcție de tip *factory* (*CreateMainWindow()*) pentru implementările obiectului *CMainWindow*.

```
#ifndef _CMAINWINDOW_H_
#define _CMAINWINDOW_H_
```

```
class CMainWindow: public CWindowImpl<CMainWindow>
public:
```

```
virtual BOOL ProcessWindowMessage
( HWND hWnd, UINT uMsg,
  WPARAM wParam, LPARAM lParam,
  LRESULT& lResult,
  DWORD dwMsgMapID) {
    BOOL bHandled = TRUE;
    switch (uMsg) {
        case WM_PAINT:
            lResult = OnPaint();
            break;
        case WM_DESTROY:
            lResult = OnDestroy();
            break;
        default:
            bHandled = FALSE;
    }
    return bHandled;
}
```

```
protected:
```

```
virtual void OnPaintImpl(HDC &hdc)=0;
```

Fereastra principală este o implementare *WindowImpl*, cu ajutorul căreia se înregistrează clasele ferestrei și manipularea mesajelor (*message handling*). Pentru a prelucra mesajele ferestrelor, implementăm *ProcessWindowMessage*. Fereastra trebuie să deseneze și operația respectivă este declarată abstract. Moștenitorii acestei clase oferă posibilitatea implementării operațiilor de desenare.

Este apelată atunci când se primește un mesaj pentru fereastră și returnează o valoare booleană, care specifică dacă mesajul a fost procesat. Implementarea reacționează la două feluri de mesaje: *WM_PAINT* și *WM_DESTROY*.

Sistemul sau alt program cer desenarea unei ferestre de aplicație.

Alte mesaje rămân neprocesate.

Desenarea este o metodă abstractă.

```

private:
    LRESULT OnPaint() {
        PAINTSTRUCT ps;
        HDC hdc=BeginPaint(&ps);
        OnPaintImpl(hdc);
        EndPaint(&ps);
        return 0;
    }

    LRESULT OnDestroy() {
        PostQuitMessage(0);
        return 0;
    }

};

CMainWindow *CreateMainWindow();
#endif

```

Pregătește fereastra pentru desenare, apelează metoda de desenare și marchează sfârșitul acesteia.

Semnalizează sistemului faptul că *thread*-ul va trebui părăsit. Bucă cu mesaje se va închela și va transmite controlul înapoi sistemului.

Declararea funcției de tip *factory* pentru implementarea *CMainWindow*.

MainWindowImpl.cpp – implementarea clasei *CMainWindow*, care desenează pătrate:

```

#include "stl.h"
#include "MainWindow.h"

class PaintInvariant {
public:
    (...)
    rHdc(refHdc), dx(0), dy(0), minL(minimumL) {
    }

    HDC& rHdc;
    int dx;
    int dy;
};

Spațiul dintre conturul stâng/drept al ferestrei și suprafața pătratului, care trebuie redesenată.

Spațiul dintre conturul sus/jos al ferestrei și suprafața pătratului, care trebuie redesenată.

```

Gruparea parametrii ce rămân constanți în timpul desenării.

Referință la un *device context* – un set pentru desenarea obiectelor grafice.

```

int minL;
};

class CQuadratWindow: public CMainWindow {
protected:
    virtual void OnPaintImpl(HDC &hdc);
private:
    void _drawRec(PaintInvariant &rPaintInvariant,
                   int x, int y, int l);
};

CMainWindow *CreateMainWindow() {
    return new CQuadratWindow();
}

void CQuadratWindow::OnPaintImpl(HDC &hdc) {
    RECT rect;
    GetClientRect(&rect);

    int diff = rect.right - rect.bottom;
    PaintInvariant paintInv(hdc, 5);
    int l;
    if(diff<0) {
        diff = -diff;
        l = rect.right/2;
        paintInv.dx = 0;
        paintInv.dy = diff/2;
    }
    else{
        l = rect.bottom/2;
        paintInv.dx = diff/2;
        paintInv.dy=0;
    }
}

```

Lungimea minimă a laturii unui pătrat.

Implementarea `CMainWindow`, care desenează pătrate.

Desenează pătrate.

Implementarea funcției de tip *factory* pentru `CMainWindow`. Furnizează o nouă instanță a clasei `CQuadratWindow`, care este o funcție declarată în header-ul `MainWindow.h`.

Implementarea metodelor de desenare.

Stabilește zona în care se poate desena.

Initializează datele invariante (variabilele care nu se vor schimba în timpul desemnării).

```

    }

    _drawRec(paintInv, l, l, l);

}

void CQuadratWindow::_drawRec(PaintInvariant &rPaintInvariant,
                                int x, int y, int l){
    if(l>rPaintInvariant.minL) {
        _drawRec(rPaintInvariant, x-1/2, y-1/2, l/2);
        _drawRec(rPaintInvariant, x-1/2, y+1/2, l/2);
        _drawRec(rPaintInvariant, x+1/2, y-1/2, l/2);
        _drawRec(rPaintInvariant, x+1/2, y+1/2, l/2);
        Rectangle(rPaintInvariant.rHdc,
                   x-1/2 + rPaintInvariant.dx,
                   y-1/2 + rPaintInvariant.dy,
                   x+1/2 + rPaintInvariant.dx,
                   y+1/2 + rPaintInvariant.dy);
    }
}

```

main.cpp:

```

#include "stdafx.h"
#include "MainWindow.h"

#include <memory>

```

Pentru std::auto_ptr - „smart pointer”

```

int APIENTRY _tWinMain (HINSTANCE hinst,
                        HINSTANCE hPrevInstance,
                        LPTSTR lpCmdLine,
                        int nCmdShow) {

```

Apelaază CreateMainWindow() și salvează pointer-ul obiectului ferestrel principale într-un smart pointer. Folosim acest tip de pointer pentru că, atunci când auto_pointeriese din scopul său, destructorul ferestrel șterge pointerul la fereastră.

```
std::auto_ptr<CMainWindow> pWndAuto(CreateMainWindow());
if(!pWndAuto.get()) {
    return -1;
}
```

Atunci când `CreateMainWindow()` nu poate să creeze o fereastră (de exemplu, deoarece nu există destulă memorie), se va returna NULL. În acest caz, aplicația se va termina.

```
pWndAuto->Create
    (0
     , CWindow::rcDefault,
     _T("Exemplu recursivitate"),
     WS_OVERLAPPEDWINDOW,
     WS_EX_CLIENTEDGE
    );
```

Nu există ferestre-părinte.

Dimensiunile default ale ferestrelor.

Numele ferestrelor.

Stilul ferestrelor.

Stil avansat al ferestrelor.

```
pWndAuto->UpdateWindow();
```

Actualizează aria client prin trimiterea unui mesaj WM_PAINT ferestrelor, atunci când zona ferestrei nu este goală.

```
pWndAuto->>ShowWindow(nCmdShow);
```

Desenază ferestra.

```
pWndAuto->UpdateWindow();
```

Actualizează aria client prin trimiterea unui mesaj WM_PAINT ferestrelor, atunci când zona ferestrei nu este goală.

```
MSG msg;
while (GetMessage(&msg, 0, 0, 0)) {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
```

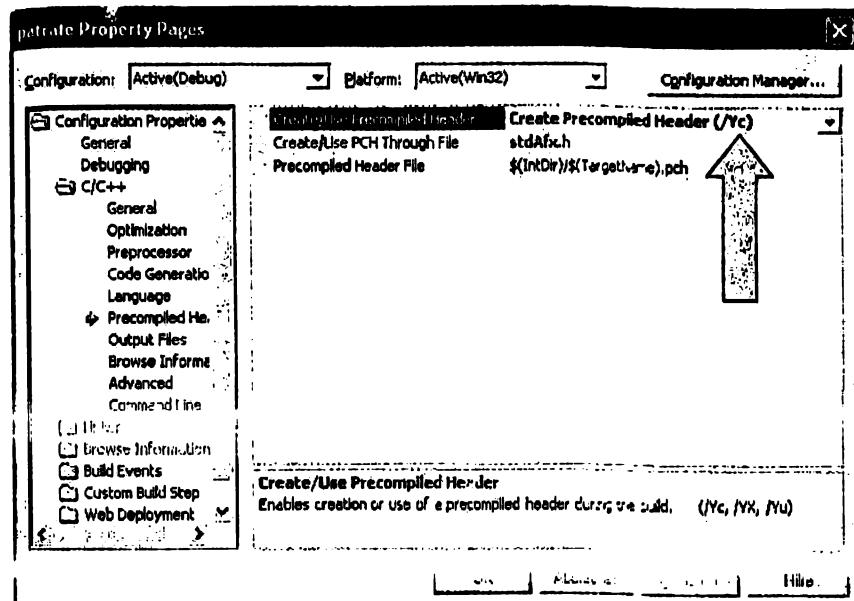
Bucă principală de mesaje: ea va fi părăsită atunci când se primește mesajul WM_DESTROY.

```
return (int) msg.wParam;
```

```
)
```

Codul de ieșire este valoarea furnizată de către PostQuitMessage: în cazul nostru, 0.

Observație. Trebuie să aveți în vedere faptul că, în meniu cu proprietățile proiectului, la secțiună Precompiled Header se va alege opțiunea Create Precompiled Header, ca în figură:



Meniul de proprietăți al proiectului

Probleme propuse

1. Modificați programul astfel încât să se deseneze figura 1: pătratele mai mici nu mai trebuie acoperite de cele mai mari).
2. Modificați programul, astfel încât, în loc de pătrate, să se deseneze triunghiuri echilaterale (figura 2). Contrairement figurei 1, triunghiul va se desenă pe un alt triunghi echilateral, a cărui latură are lungimea jumătății unei laturi triunghiului initial.

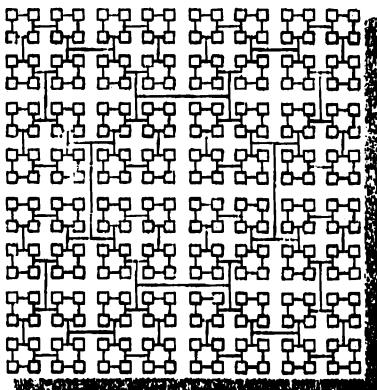


Figura 1

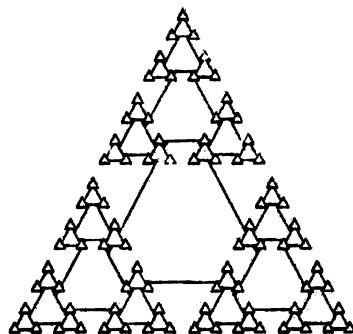


Figura 2

3. Dacă programăți în Windows, implementați programul fără a utiliza biblioteca ATL, ci facilitățile Windows standard.

Problema 11. Pătrate și cercuri (recursivitate indirectă)

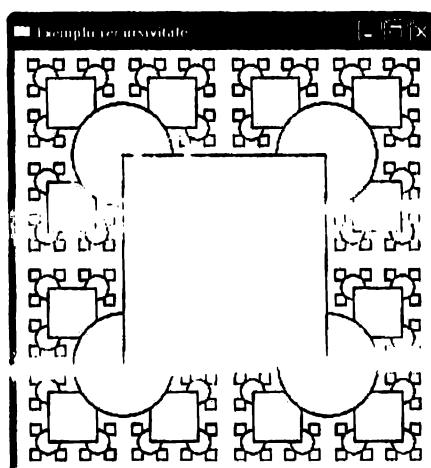
Scrieți un program recursiv, care să deseneze figura alăturată.

Analiza problemei și proiectarea

scrierii

În vârfurile unui pătrat cu lungimea laturii L , se desenază cercuri cu diametrul $L/2$, centrate în vârfurile pătratului. În următoarele etape, în loc de cercuri sunt desenate pătrate centrate

în vârfurile lor, astfel încât cercurile sunt înscrise în pătrate, care sunt invizibile și în vârfurile lor se vor desena corespunzător pătrate centrate cu laturile înjurătășite. Procesul se va relua până când latura unui pătrat ajunge mai mică decât 10. Prezentăm mai jos o implementare DOS simplă, care este compilată cu Borland C++ 3.1:



Program

```
#include <graphics.h>
#include <iostream.h>
#include <stdlib.h>

void initGrafix()
{
    int gdet = DETECT, gm, err;
    initgraph(&gdet, &gm, "c:\\borlandc\\BGI");
    err = graphresult();
    if(err != grOk)
    {
        cout<< "Grafic error: "<< grapherrmsg(err)<< endl;
        cout<< "Press any key to quit." ;
        exit(1);
    }
}

void drawSquare(int x, int y, int l);
void drawCircle(int x, int y, int r)
{
    if(r>25)
    {
        circle(x, y, r);
        drawSquare(x, y, r);
    }
}

void drawSquare (int x, int y, int l)
{
    if(l>10)
    {
        rectangle(x-1/2, y-1/2, x+1/2, y+1/2);
        drawCircle (x-1/2, y-1/2, 1/2);
        drawCircle (x-1/2, y+1/2, 1/2);
        drawCircle (x+1/2, y-1/2, 1/2);
        drawCircle (x+1/2, y+1/2, 1/2);
    }
}

int main()
{
    initGrafix();
    setcolor(0);
    setlinestyle(0, 1, 2);
```

```

    setfillstyle(1, 15);
    bar(0, 0, getmaxx(), getmaxy());
    drawSquare(getmaxx()/2, getmaxy()/2, getmaxy()/2);
    closegraph();
    return 0;
}

```

Prezentăm în continuare o implementare Windows, care seamănă cu implementarea ATL de la problema anterioară. Se creează un proiect cu fișierele *stdafx.h*, *MainWindow.h* și *main.cpp*, care coincid cu cele de mai sus. Ceea ce se schimbă este doar implementarea *MainWindowImpl.cpp*, după cum urmează:

```

#include "stdafx.h"
#include "MainWindow.h"

class PaintInvariant{
public:
    PaintInvariant(HDC& refHdc, int minimumL = 10):
        rHdc(refHdc), dx(0), dy(0), minL(minimumL) {}

    HDC& rHdc;

    int dx;

    int dy;

    int minL;
};

class CSquareCircleWindow: public CMainWindow{
protected:
    virtual void OnPaintImpl(HDC &hdc);

private:
    void _drawRec(PaintInvariant &rPaintInvariant,
                   int x, int y, int l);
    void _drawCircle(PaintInvariant &rPaintInvariant,
                      int x, int y, int l);
};

CMainWindow *CreateMainWindow() {
    return new CSquareCircleWindow();
}

```

```
void CSquareCircleWindow::OnPaintImpl(HDC &hdc) {
    RECT rect;
    GetClientRect(&rect);

    int diff = rect.right - rect.bottom;
    PaintInvariant paintInv(hdc, 10);
    int l;
    if (diff < 0) {
        diff *= -1;
        l = rect.right / 2;
        paintInv.dx = 0;
        paintInv.dy = diff / 2;
    }
    else{
        l = rect.bottom / 2;
        paintInv.dx = diff / 2;
        paintInv.dy = 0;
    }

    _drawRec(paintInv, l, l, l);
}

void CSquareCircleWindow::_drawRec(
    PaintInvariant &rPaintInvariant,
    int x, int y, int l)
{
    if (l > rPaintInvariant.minL) {
        _drawCircle(rPaintInvariant, x - l / 2, y - l / 2, l / 2);
        _drawCircle(rPaintInvariant, x - l / 2, y + l / 2, l / 2);
        _drawCircle(rPaintInvariant, x + l / 2, y - l / 2, l / 2);
        _drawCircle(rPaintInvariant, x + l / 2, y + l / 2, l / 2);
        Rectangle(rPaintInvariant.hDC
            , x - l / 2 + rPaintInvariant.dx,
            y - l / 2 + rPaintInvariant.dy,
            x + l / 2 + rPaintInvariant.dx,
            y + l / 2 + rPaintInvariant.dy);
    }
}

void CSquareCircleWindow::_drawCircle(
    PaintInvariant &rPaintInvariant,
    int x, int y, int l)
{
    if (l > rPaintInvariant.minL) {
        _drawRec(rPaintInvariant, x - l / 2, y - l / 2, l / 2);
        _drawRec(rPaintInvariant, x - l / 2, y + l / 2, l / 2);
```

```

    _drawRec(rPaintInvariant, x+1/2, y-1/2, 1/2);
    _drawRec(rPaintInvariant, x+1/2, y+1/2, 1/2);
    Ellipse(rPaintInvariant.rHdc,
        x-1/2 + rPaintInvariant.dx,
        y-1/2 + rPaintInvariant.dy,
        x+1/2 + rPaintInvariant.dx,
        y+1/2 + rPaintInvariant.dy);
    }
}

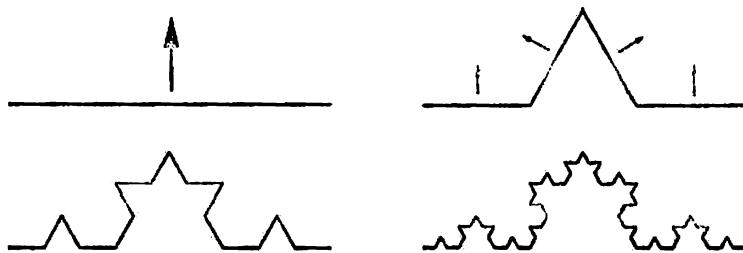
```

Probleme propuse

1. Calculați suma ariilor vizibile ale tuturor cercurilor.
2. Implementați un program care folosește de asemenea recursivitatea indirectă astfel:
 - *drawSquare* apelează de patru ori metoda *drawCircle* (desenează patru cercuri de diametru $L/2$, ale căror centre sunt vârfurile pătratului);
 - *drawCircle* apelează o dată *drawTriangle* (desenează triunghiul echilateral circumscris cercului);
 - *drawTriangle* apelează de trei ori *drawSquare* (desenează trei pătrate cu latura jumătate din cea a triunghiului, centrate în vârfurile sale).

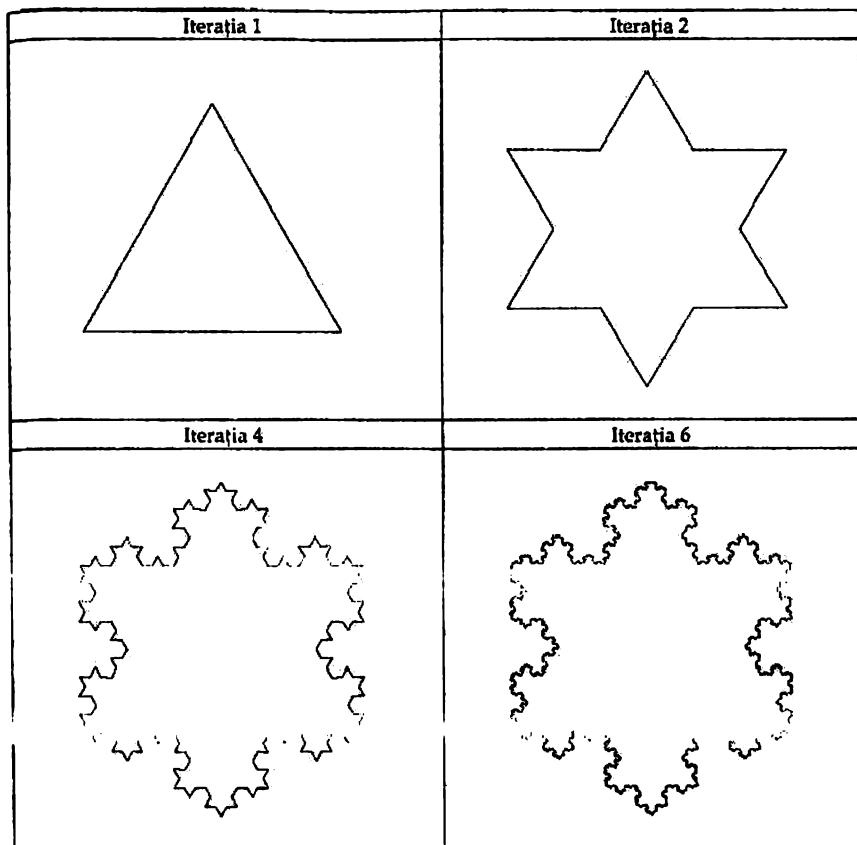
Problema 12. Curba fulgului de zăpadă a lui Koch

A fost descrisă în 1904 de către matematicianul suedez Helge von Koch (1870-1924) și este numită după numele fondatorului său și formator. Curba lui Koch se bazează pe un proces construit iterativ: la început, este formată numai dintr-un segment și la fiecare iterație fiecare segment al curbei se transformă prin „construcția în triunghi echilateral” (segmentul se împarte în trei părți egale, cea din mijloc se sterge și se înlocuiește cu alte două segmente, ce formează cu cea ștersă un triunghi echilateral).

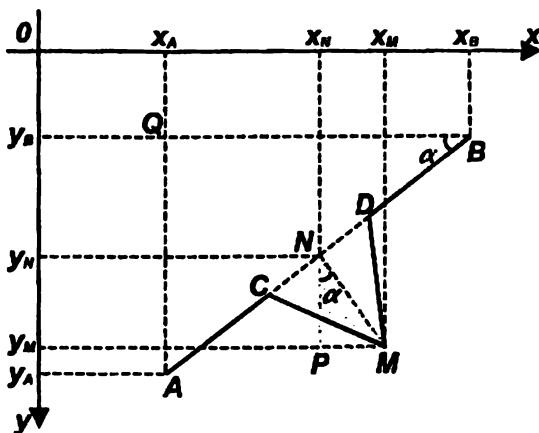


Algoritmul de obținere a curbei lui Koch

Curba fulgului de zăpadă se creează atunci când procesul iterativ începe cu trei segmente care formează un triunghi echilateral. Scrieți un program care desenează fulgul de zăpadă pentru o iterare dată. Exemplu (fișier de intrare *iteration.in*):



Analiza problemei și proiectarea soluției



Corespunzător punctelor A și B în plan, cu coordonatele $A(x_A, y_A)$ și $B(x_B, y_B)$, trebuie să găsim coordonatele punctelor C , D și M cu proprietățile: $C, D \in$ dreptei (AB) , $AC = CD = DB = CM = MD$.

Notăm mijlocul segmentului AB cu $N(x_N, y_N)$. Coordonatele acestui punct sunt:

$$x_N = \frac{x_A + x_B}{2}, \quad y_N = \frac{y_A + y_B}{2}.$$

Mai departe, calculăm coordonatele punctului C . Au loc:

$$\frac{x_C - x_A}{x_D - x_A} = \frac{1}{3} \Rightarrow 3x_C - 3x_A = x_B - x_A \Rightarrow x_C = \frac{2x_A + x_B}{3},$$

$$\frac{y_C - y_A}{y_B - y_A} = \frac{1}{3} \Rightarrow 3y_C - 3y_A = y_B - y_A \Rightarrow y_C = \frac{y_A + 2y_B}{3}.$$

Pe această cale se ajunge la: $x_D = \frac{x_A + 2x_B}{3}$ și $y_D = \frac{y_A + 2y_B}{3}$.

Deoarece MCD este un triunghi echilateral cu lungimea laturii $\frac{AB}{3}$ și înălțimea MN , rezultă:

$$MN = CM \cdot \sin \frac{\pi}{3} = \frac{AB}{3} \cdot \frac{\sqrt{3}}{2} = \frac{\sqrt{3}}{6} \cdot AB.$$

Observăm că $\alpha = \angle PNM = \angle QBA$ (PN este perpendiculară pe QB și NM perpendiculară pe AB: $PN \perp QB$ și $NM \perp AB$). În triunghiul ABQ avem:

$$\sin \alpha = \frac{y_A - y_B}{AB} \text{ și } \cos \alpha = \frac{x_B - x_A}{AB}.$$

Și le folosim în triunghiul NMP:

$$dx = MN \cdot \sin \alpha = \frac{\sqrt{3}}{6} \cdot AB \cdot \frac{y_A - y_B}{AB} = \frac{\sqrt{3}}{6} \cdot (y_A - y_B)$$

$$dy = MN \cdot \cos \alpha = \frac{\sqrt{3}}{6} \cdot AB \cdot \frac{x_B - x_A}{AB} = \frac{\sqrt{3}}{6} \cdot (x_B - x_A).$$

În continuare, rezultă coordonatele punctului M:

$$x_M = x_N + dx = \frac{x_A + x_B}{2} + \frac{\sqrt{3}}{6} \cdot (y_A - y_B)$$

$$y_M = y_N + dy = \frac{y_A + y_B}{2} + \frac{\sqrt{3}}{6} \cdot (x_B - x_A).$$

Această soluție a fost prezentată în [Ran99].

În continuare se va crea un proiect în C++ Windows, folosindu-se tot cele trei acțiuni grafice precedente (problemele 10 și 11). Se va crea un proiect cu fișierele *stdafx.h*, *MainWindow.h* și *main.cpp*, identice cu cele de la problema 10. Se va modifica doar implementarea *MainWindowImpl.cpp*, ca mai jos.

```
#include "stdafx.h"
#include "MainWindow.h"

// Incluzări <fisiere sau>
#include <cmath>

class PaintInvariant {
public:
    PaintInvariant(HDC refHdc, int minimumL = 10) :
        rHdc(refHdc), dx(0), dy(0), minL(minimumL) {}

    void drawLine(HDC hdc, int x1, int y1, int x2, int y2);
    void drawCircle(HDC hdc, int cx, int cy, int radius);
    void drawTriangle(HDC hdc, int x1, int y1, int x2, int y2, int x3, int y3);
};
```

```

HDC& rHdc;
int dx;
int dy;
int minL;
};

class CKochWindow: public CMainWindow {
protected:
    virtual void OnPaintImpl(HDC &hdc);
private:
    void _drawKoch(PaintInvariant &rPaintInvariant, int n,
        int xa, int ya, int xb, int yb);
};

CMainWindow *CreateMainWindow() {
    return new CKochWindow();
}

void CKochWindow::OnPaintImpl(HDC &hdc) {

    RECT rect;
    GetClientRect(&rect);

    int diff = rect.right - rect.bottom;
    PaintInvariant paintInv(hdc, 5);
    int l;
    if(diff<0) {
        l=0;
        t=rect.right/2;
        paintInv.dx = 0;
        paintInv.dy = diff/2;
    }
    else{
        l=1;
        t=(rect.right+rect.bottom)/2;
        paintInv.dx = diff/2;
        paintInv.dy=0;
    }

    ...  

    std::ifstream in("iteration.in");
    if(in && !in.eof()){
        in>>n;
        double xa, ya, xb, yb, t=l;
        double aux=sqrt(3.0);
        xa=l; ya=l-t;
        xb=l-aux*t/2; yb=l+t/2;
        _drawKoch(paintInv, n,
            (int)xa, (int)ya, (int)xb, (int)yb);
    }
}

```

```

xa=1+aux*t/2; ya=1+t/2;
_drawKoch(paintInv, n,
           (int)xb, (int)yb, (int)xa, (int)ya);
xb=1; yb=1-t;
_drawKoch(paintInv, n,
           (int)xa, (int)ya, (int)xb, (int)yb);
}

}

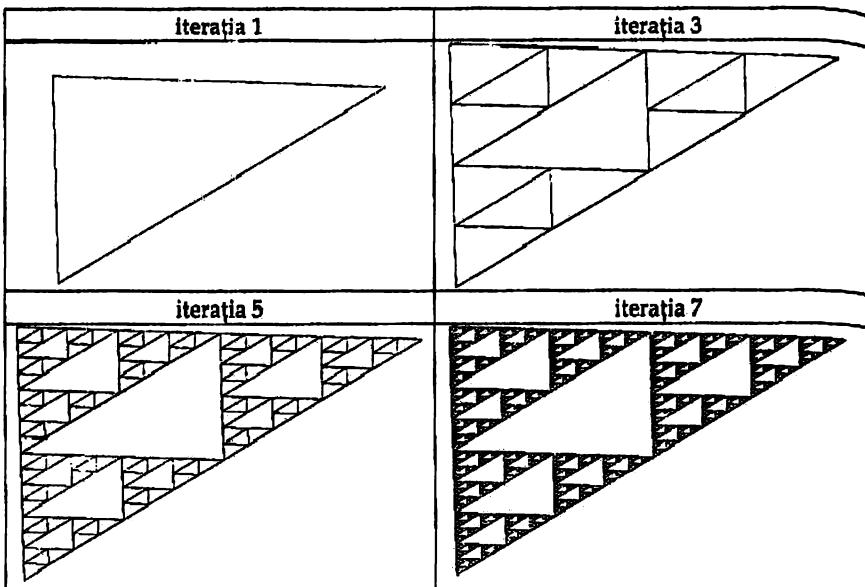
void CKochWindow::drawKoch(PaintInvariant &rPaintInvariant,
                           int n, int xa, int ya, int xb, int yb){
    double r=sqrt(3.0)/6.0;
    if(n>1){
        double xc, yc, xm, ym, xd, yd;
        xm=(xa+xb)/2+r*(ya-yb);
        ym=(ya+yb)/2+r*(xb-xa);
        xc=(2*x+a+b)/3; yc=(2*y+a+b)/3;
        xd=(xa+2*xb)/3; yd=(ya+2*yb)/3;
        xM =  $\frac{x_A + x_B}{2} + \frac{\sqrt{3}}{6} \cdot (y_A - y_B)$ 
        yM =  $\frac{y_A + y_B}{2} + \frac{\sqrt{3}}{6} \cdot (x_B - x_A)$ 
        xC =  $\frac{2x_A + x_B}{3}$ , yC =  $\frac{2y_A + y_B}{3}$ 
        xD =  $\frac{x_A + 2x_B}{3}$ , yD =  $\frac{y_A + 2y_B}{3}$ 
        _drawKoch(rPaintInvariant, n-1,
                   (int)xa, (int)ya, (int)xc, (int)yc);
        _drawKoch(rPaintInvariant, n-1,
                   (int)xc, (int)yc, (int)xm, (int)ym);
        _drawKoch(rPaintInvariant, n-1,
                   (int)xm, (int)ym, (int)xd, (int)yd);
        drawKoch(rPaintInvariant, n-1,
                  (int)xd, (int)yd, (int)xb, (int)yb);
    } else{
        POINT p[2];
        p[0].x=xa; p[0].y=ya;
        Desenează segmentul AB. Altă posibilitate:
        MoveToEx(rPaintInvariant.rHdc,
                  xa, ya, NULL);
        Polygon(rPaintInvariant.rHdc, p, 2);
    }
}

```

Probleme propuse

1. Calculați aria pe care o ocupă curba fulgului de zăpadă la o iterație dată atunci când triunghiul inițial are lungimea laturii L . Care sunt lungimea curbeli și aria pentru număr de iterări care tind la infinit?
2. Triunghiul lui Sierpinski. Se consideră un triunghi oarecare, în care se trasează cele trei linii mijlocii și se elimină triunghiul care se formează în interior. Acest

procedeu se repetă pentru triunghiurile formate, iar figura obținută poartă numele de *triunghiul lui Sierpinski*. Scrieți un program care desenează triunghiul lui Sierpinski pentru o iterare dată. Exemplu:



Calculați aria triunghiului lui Sierpinski pentru o iterare dată, atunci când triunghiul inițial are aria A . Determinați această arie atunci când numărul iterației tinde la infinit.

3. Fractal space-filling. Fie mulțimea $P(0, 1) = \{(x, y) \mid 0 < x, y < 1\}$, care este mulțimea tuturor punctelor din pătratul unitate cu vârfurile $(0, 0), (1, 0), (1, 1)$ și $(0, 1)$ [Bar93] este prezentată și demonstrată o metodă care „umple” acest pătrat. Se consideră următoarele tablouri:

Index	a	b	c	d	e	f
1	0	0.5	0.5	0	0	0
2	0.5	0	0	0.5	0	0.5
3	0.5	0	0	0.5	0.5	0.5
4	0	-0.5	-0.5	0	1	0.5

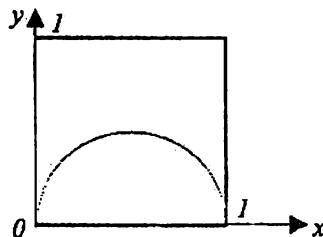
Fie K_0 o curbă în $P(0, 1)$. Pentru toate numerele naturale $i \geq 0$, se creează curba K_{i+1} pe baza curbei K_i , prin înlocuirea fiecărui punct din K_i cu următoarele patru puncte:

$$w_n \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a_n & b_n \\ c_n & d_n \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e_n \\ f_n \end{bmatrix} \text{ pentru } n = 1, \dots, 4.$$

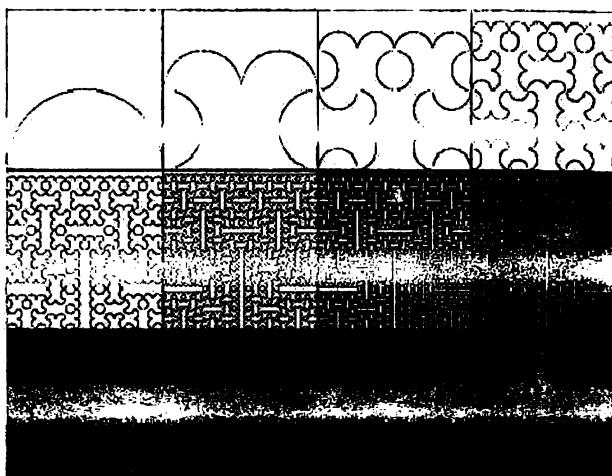
Au loc deci: $\begin{cases} x_{n,i+1} = a_n \cdot x_i + b_n \cdot y_i + e_n \\ y_{n,i+1} = c_n \cdot x_i + d_n \cdot y_i + f_n \end{cases}$, pentru $n = 1, \dots, 4$. Formal, putem scrie

pentru orice $i \geq 0$: $K_{i+1} = \bigcup_{n=1}^4 \{(x_{n,i+1}, y_{n,i+1}) | (x_i, y_i) \in K_i\}$.

Pentru curba inițială K_0 :



$K_0, K_1, K_2, \dots, K_n$



Scrieti un program care creeaza diverse curbe initiale K și desenează în această formă primele 12 iterații.

Nuță: o rezolvare în Borland C pentru DOS a problemei, folosind operatorii pe biți, se găsește în [Log06], p. 394.



Lebede pe râul Isar în München

Capitolul 9

Divide et Impera

7 probleme complet rezolvate, 18 probleme propuse

- Fundamente
- Cel mai mare divizor comun
- Turnurile din Hanoi
- Integrală cu regula trapezului
- *QuickSort* (sortarea rapidă)
- *MergeSort* (sortarea prin interclasare)
- Arboi Quad (Quad-Trees)
- Transformata Fourier discretă (DFT)

O fi bine, o fi rău, uneori este foarte plăcut să spargi ceva.

F.M. Dostoievski

Fundamente

Se presupune că formularea *divide et impera* îi aparține regelui francez Ludovic al XI-lea (1461-1483). Unii istorici însă î-o atribuie și lui Iulius Cezar (100 î.Ch.-44 î.Ch.). Ea ar trebui să însemne că adversarii trebuie împărăști și slăbiți prin lupte între ei, pentru a putea fi învinși și supuși.

Vom considera mai departe însă această formulare dintr-o perspectivă mult mai civilizată. Îndemnul trebuie să ne ajute la soluționarea unor probleme prin împărțirea lor în subprobleme de dimensiune mai mică, rezolvarea lor și combinarea rezultatelor în scopul obținerii soluției pentru problema inițială. La rândul lor, subproblemele se pot împărti în probleme asemănătoare de dimensiuni mai mici, care se rezolvă cu aceeași metodă. Această manevră se reia recursiv, până când subproblemele obținute devin elementare și pot fi rezolvate direct.

Pașii rezolvării unei probleme folosind principiul *Divide et Impera* sunt:

- *Divide*: împarte problema recursiv în subprobleme, până când au dimensiuni elementare;
- *Impera*: rezolvă suhproblemele;
- *Impera*: combină soluțiile subproblemelor pentru obținerea soluției problemei inițiale.

Forma generală a algoritmului:

```
ALGORITM_DivideEtImpera(P, n, S)
If (n≤m) Then
    ...
End_If
Else Execute
    Divide (P, n) in (P1, m1), (P2, m2), ..., (Pk, mk)
    DivideEtImpera(P1, m1, S1)
    DivideEtImpera(P2, m2, S2)
    ...
    DivideEtImpera(Pk, mk, Sk)
    Combine S1, S2, ..., Sk to obtain S
End_Else
END_ALGORITM_DivideEtImpera(P, n, S)
```

Exemple cunoscute pentru aplicarea metodei sunt: problema turnurilor din Hanoi, QuickSort, MergeSort.

Problema 1. Cel mai mare divizor comun

În fișierul *numere.in* se găsesc mai multe numere naturale (maxim 2000), separate prin spații. Scrieți o metodă care să calculeze cel mai mare divizor comun al acestora. Atunci când fișierul de intrare nu există sau nu conține date, scrieți 0 în fișierul de ieșire.

numere.in	ecran
45 60 125 345 65 9875 4555	cmmdc = 5

Analiza problemei și proiectarea soluției

Definim tipul *TVector* pentru a stoca numerele citite din fișierul de intrare. Este evident că putem calcula cel mai mare divizor comun pentru prima jumătate a vectorului și pentru a doua jumătate, iar soluția problemei este cel mai mare divizor comun al celor două. Condiția de terminare este atinsă atunci când numărul elementelor devine 1 sau 0.

Program

```
#include <iostream>
#include <vector>

typedef std::vector<unsigned> TVector;
unsigned cmmdc(unsigned a, unsigned b) {
    unsigned r;
    while(b) {
        r = a % b;
        a = b;
        b = r;
    }
    return a;
}

unsigned Divide_et_Impera(TVector& a, int iMin, int iMax) {
    if(iMin > iMax) {
        int middle = (iMin+iMax)/2;
        int d1 = Divide_et_Impera(a, iMin, middle);
        int d2 = Divide_et_Impera(a, middle + 1, iMax);
        if(d1 <= d2)
            return d1;
        else
            return d2;
    }
    else
        return cmmdc(a[iMin], Divide_et_Impera(a, iMin+1, iMax));
}
```

1. $a_1 \leftarrow a, b_1 \leftarrow b, i \leftarrow 1$
 2. While ($b_i \neq 0$) Do
 2.1. $a_{i+1} \leftarrow b_i$
 2.2. $b_{i+1} \leftarrow r_i (=a_i \bmod b_i)$
 2.3. $i \leftarrow i+1$
 End_While
 3. $cmmdc(a, b) = a_1$

```

    return cmmdc(d1, d2);
}
if(0<=iMin && iMin< a.size()) return a[iMin];
else return 0;
}

int main(){
    std::ifstream f("numere.in");
    unsigned i, n = 0;
    TVector a;
    while(f && !f.eof() && f>>i) a.push_back(i);
    std::cout << "cmmdc = " << Divide_et_Impera(a, 0, a.size()-1);
    return 0;
}

```

Probleme propuse

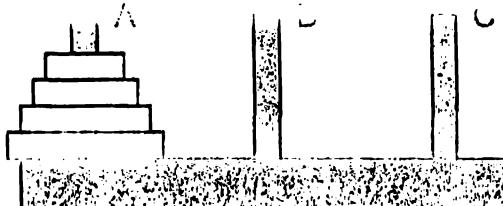
1. Scrieți un program care să folosească aceeași metodă pentru a calcula cel mai mare și cel mai mic dintre numerele date.
2. Scrieți un program care, prin aceeași metodă, să caute un număr dat în fișierul *numere.in*.

Problema 2. Turnurile din Hanoi

Se presupune că unul dintre cele mai populare jocuri din toate timpurile, turnurile din Hanoi, a fost inventat în 1800 de către fratele lui Clément Lecocq. Acesta este unul dintre exemplele standard pentru metoda *Divide et Impera* în literatura de specialitate. Jocul este format din trei tije, A, B și C, pe care se introduc unul sau mai multe discuri. La început, toate discurile se află pe tija A, în ordinea descrescătoare a diametrelor, cel mai mare disc fiind la baza tijei. Scopul jocului este mutarea tuturor discurilor de pe

tija A pe tija C, folosind doar tijele B și C și urmând oricare pe zi, regula jocului:

- la un moment dat, se mută doar un disc de pe o tijă pe alta;
- nu este permisă amplasarea unui disc de dimensiune mai mare peste un disc de dimensiune mai mică.



Găsiți pentru un număr natural n , citit de la tastatură, cea mai scurtă secvență de mutări ce rezolvă problema pentru n discuri inițiale. Mutările se vor scrie în fișierul *hanoi.out*, ca în exemplu:

Tastatură	<i>hanoi.out</i>
3	(A, C) (A, B) (C, B) (A, C) (B, A) (B, C) (A, C)

Analiza problemei și proiectarea soluției

Soluția se construiește pornind de la următoarea observație: pentru a muta n discuri de pe tija *A* pe tija *C* cu ajutorul tijei *B*, vom muta mai întâi $n - 1$ discuri de pe tija *A* pe tija *B* prin intermediul tijei *C*, apoi executăm mutarea *A* \rightarrow *C*, urmată de mutarea a $n - 1$ discuri de pe tija *B* pe tija *C* prin intermediul tijei *A*, adică:

$$Hanoi(n, A, C, B) = \begin{cases} A \rightarrow C, \text{ dacă } n=1 \\ Hanoi(n-1, A, B, C), A \rightarrow C, Hanoi(n-1, B, C, A), \text{ altfel.} \end{cases}$$

Program

```
#include <fstream>
#include <iostream>

std::ofstream f("hanoi.out");

void write(char A, char B){
    f << "(" << A << "," << B << ")";
}

void hanoi(int n, char A, char C, char B){
    if (1==n) write(A, C);
    else{
        Hanoi(n-1, A, B, C);
        write(B, C);
        Hanoi(n-1, B, C, A);
    }
}

int main()
{
    int n;
    std::cout << " n = ";
    std::cin >> n;
    hanoi(n, 'A', 'C', 'B');
    return 0;
}
```

Probleme propuse

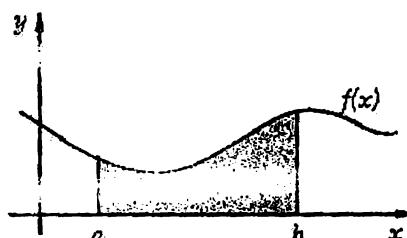
- Demonstrați că numărul total de mutări este $2^n - 1$.
- Scrieți un program care simulează grafic într-o formă cât mai frumoasă și mai eloventă mutarea discurilor.
- Implementați un program nerecursiv pentru rezolvarea problemei: la fiecare mutare impară, se va muta cel mai mic disc (1) de pe tija pe care se află pe următoarea în succesiunea circulară de tije ABCABCABC...; pentru mutările pare, se execută mutarea posibilă care nu implică discul 1.
- Numărul minim de mutări necesare aplicând algoritmul anterior este $2^n - 1$. Pentru două numere naturale n ($0 \leq n \leq 100$) și m ($0 \leq m \leq 2^n - 1$), determinați a m -a mutare și configurația tijelor din acel moment.

Problema 3. Integrală cu regula trapezului

Integrala unei funcții în plan este aria suprafeței care se formează între graficul funcției și axa Ox. Atunci când funcția are două variabile, integrala reprezintă un volum. Regula trapezului este o metodă matematică cunoscută pentru aproximarea numerică a integralei unei funcții pe intervalul $[a, b]$ (suprafața gri din figura alăturată). În zona acoperită de funcție în intervalul dat se construiește un trapez și se calculează aria acestuia. Pentru a aproxima mai bine aria, intervalul dat se împarte în mai multe intervale mai mici și aria dintre curba funcției și axa Ox se apropiie de suma ariilor trapezelor. Calculați valoarea $\int_a^b f(x) dx$ folosind metoda trapezelor, astfel încât înălțimea fiecărui trapez a cărui aria se sumează să fie mai mică decât ϵ (epsilon) = 0,0001.

Date de intrare. În fișierul `integrala.in` se găsesc mai multe perechi (a, b) de numere reale, unde $a < b$, $a \geq -100$, $b \leq 200$, $a \leq b$, ce reprezintă intervale.

Date de ieșire. Pentru fiecare interval din fișierul de intrare, scrieți câte o linie în fișierul de ieșire cu valoarea integrală, în exemplu:



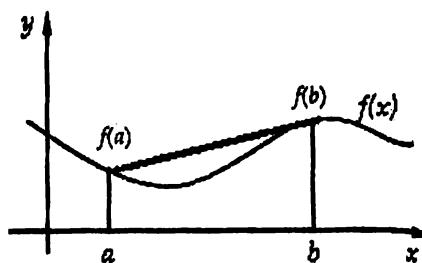
integrala.in	integrala.out
0 1	$I(0, 1) = 0.785393$
90.09 123.45	$I(90.09, 123.45) = 0.00299929$
-45.6 23	$I(-45.6, 23) = 3.07615$

Analiza problemei și proiectarea soluției

Aria trapezului cu vârfurile de coordinate $(a, 0)$, $(b, 0)$, $(a, f(a))$ și $(b, f(b))$ este:

$$(b-a) \left(\frac{f(a)+f(b)}{2} \right)$$
.

Cu ajutorul valorii *epsilon*, determinăm o condiție de oprire pentru algoritmul recursiv, și anume atunci când lungimea intervalului devine mai mică sau egală cu *epsilon*. În acest caz, integrala se va aproxima cu aria trapezului.



Program

```
#include <iostream>

const double eps=0.0001;

using namespace std;

double f(double x){
    return 1/(1+x*x);
}

double integral(double a, double b){
    if(b-a>eps){
        double c = (a+b)/2;
        return integral(a,c) + integral(c, b);
    } else{
        return (b-a)*(f(b)+f(a))/2;
    }
}

int main(){
    double a, b;
    ifstream in("integrala.in");
    ofstream out("integrala.out");
    while (!in.eof() && in>>a>>b){
        out << "I(" << a << ", " << b << ") = ";
        out << integral(a, b) << " " << endl;
    }
    return 0;
}
```

Probleme propuse

1. Scrieți o variantă iterativă pentru problemă.
2. Modificați programul astfel încât să folosiți metoda dreptunghiului, care spune că zona dintre graficul funcției și axa Ox pe intervalul $[a, b]$ se poate approxima cu aria dreptunghiului cu o latură $|f((a+b)/2)|$ și cealaltă lungimea intervalului.

Problema 4. QuickSort (sortarea rapidă)

În fișierul *QSort.in* se găsesc mai multe numere naturale. Implementați metoda QuickSort pentru a le sorta și scrieți rezultatul în fișierul *QSort.out*. Exemplu:

<i>QSort.in</i>	<i>QSort.out</i>
345 67 13 789 90 76 -45	-45 13 67 76 90 345 789

Analiza problemei și proiectarea soluției

QuickSort este un algoritm de sortare, ce a fost prezentat pentru prima dată de către C.A.R. Hoare (n. 1934) în 1962. QuickSort este în medie foarte rapid atunci când trebuie sortate mulțimi mari de date. Algoritmul alege un element aleator, din secvența de sortat, care reprezintă pivotul (punct de deviație). Apoi, secvența de date se împarte în două zone: una în care elementele sunt mai mici sau egale cu pivotul și una în care elementele sunt mai mari (în cadrul zonelor respective, elementele nu sunt sorteate). Apoi, fiecare zonă se împarte recursiv în două zone, pe baza unui pivot corespunzător, până când se ajunge la zonă cu un element (problema elementară). QuickSort este, în medie, cel mai rapid algoritm de sortare.

În implementarea noastră, alegem ca pivot primul element din secvența $a[inf]$, $a[inf+1]$, ..., $a[sup]$ și folosim variabilele contor i și j , cu care ne deplasăm spre dreapta, respectiv spre stânga în secvența dată. După împărțirea datelor în două zone (rezulta din ceea ce urmărește să fie realizată), dacă avem două cu elemente mai mari decât pivotul, vom plasa elementul pivot între ele și vom apela recursiv metoda *quickSort()* pentru fiecare zonă. Condiția de terminare este ca dimensiunea secvenței să fie 1 ($inf=sup$).

Complexitatea. Dacă distribuția conduce, în general la împărțirea în două zone, ale căror dimensiuni sunt aproape egale, atunci complexitatea algoritmului este $O(n \log n)$. Cel mai nefavorabil caz se produce atunci când cele două zone obținute au lungimi foarte diferite, complexitatea algoritmului devenind pătratică.

Program

```
#include <iostream>
#include <vector>

using namespace std;

void readData(vector<int> &a) {
    int aux;
    ifstream f("QSort.in");
    while(f && !f.eof() && f>>aux)
        a.push_back(aux);
}

void quickSort(vector<int> &a, int inf, int sup){
    if(inf<sup){
        int pivot = a[inf], aux;
        int i = inf + 1, j = sup;
        while(i<=j){
            while(i<=sup && a[i] <= pivot) i++;
            while(j>=inf && a[j] > pivot) j--;
            if(i<j && i<=sup && j>=inf){
                aux = a[i];
                a[i] = a[j];
                a[j] = aux;
                i++; j--;
            }
        }
        a[inf] = a[i]; a[i] = pivot;
        quickSort(a, inf, i-1);
        quickSort(a, i+1, sup);
    }
}

void write(vector<int> a){
    ofstream f("QSort.out");
    for(int i=0; i<a.size(); i++) f << a[i] << " ";
}

int main(){
    vector<int> a;
    readData(a);
    quickSort(a, 0, a.size()-1);
    write(a);
    return 0;
}
```

Probleme propuse

1. Scrieți pe hârtie pașii algoritmului pentru exemplul dat (*QSort.in*).
2. Modificați programul de mai sus, astfel încât elementul pivot să fie ales întâmplător din intervalul dat.
3. Se consideră un vector cu n elemente. Scrieți un algoritm care returnează al k -lea element ($0 \leq k < n$) al vectorului virtual sortat, fără a sorta însă vectorul dat.

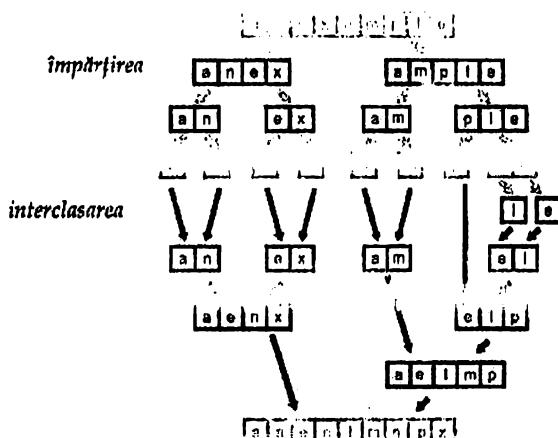
Problema 5. MergeSort (sortarea prin interclasare)

În fișierul *MSort.in* se găsesc mai multe numere naturale. Implementați metoda *MergeSort*, cu ajutorul căreia numerele date să fie ordonate crescător. Exemplu:

MSort.in	MSort.out
345 67 13 789 90 76 -45	-45 13 67 76 90 345 789

Analiza problemei și proiectarea soluției

MergeSort este, ca și *QuickSort*, o metodă eficientă de sortare. Ea procedează astfel: vectorul dat se împarte în două părți, care vor fi ordonate recursiv și apoi interclasate, pentru a obține vectorul inițial sortat. Complexitatea procesului de interclasare este liniară, pentru că ambele secvențe sunt deja sortate. Complexitatea algoritmului este $O(n \log n)$, iar, în practică, timpul de execuție este în medie mai mare decât cel obținut cu metoda *QuickSort*.



Program

```
#include <iostream>
#include <vector>

using namespace std;
template <class T>

void mergeSort(T& v) {
    if (v.size() <= 1) return;
    T v1(v.size() / 2);
    T v2(v.size() - v1.size());
    for (int i = 0; i < v1.size(); i++) v1[i] = v[i];
    for (int i = 0; i < v2.size(); i++) v2[i] = v[i + v1.size()];
    mergeSort(v1);
    mergeSort(v2);
    int i1 = 0, i2 = 0;
    for (int k = 0; k < v1.size() + v2.size(); k++) {
        if ((i1 >= v2.size()) || (i1 < v1.size() && v1[i1] <= v2[i2])) {
            v[k] = v1[i1]; i1++;
        }
        else {
            v[k] = v2[i2]; i2++;
        }
    }
}

int main() {
    vector<int> v;
    int aux;
    ifstream in("MSort.in");
    ofstream out("MSort.out");
    while (in && !in.eof() && in >> aux) v.push_back(aux);
    mergeSort(v);
    for (int i = 0; i < v.size(); i++) cout << v[i] << " ";
    return 0;
}
```

Probleme propuse

- Modificați programul astfel încât metoda `mergeSort()` să aibă trei parametri: Vector `v`, `int inf` și `int sup`, cu semnificațiile că `inf` și `sup` sunt capetele secvenței ce trebuie ordonate în vector.

```
// void mergeSort (Vector v, int inf, int sup) (...)
```

- Implementați o reprezentare grafică adecvată pentru algoritmul `MergeSort`.

Problema 6. Arbori Quad (Quad-Trees)

Arborii *Quad* sunt arbori ale căror noduri interioare au maxim patru succesi și sunt utilizati, de exemplu, pentru prelucrarea fotografiilor. Arborii *Quad* de tip *point region* (*point region Quad-Trees*) sunt arbori *Quad* de un tip special, în care un nod are patru succesi nici unul. În această problemă, vom utiliza astfel de arbori *Quad*.

Idea de bază este aceea că fiecare imagine se poate împărți în patru zone egale, corespunzător celor patru cadrane. Fiecare cadran poate fi, la rândul său, împărțit în patru zone egale și aşa mai departe. Într-un arbore *Quad* de tip *point region*, imaginea este reprezentată de un nod-tată, în timp ce cadranele sunt reprezentate ca fiind cele patru noduri-fii (într-o ordine dată). Dacă imaginea are o singură culoare, atunci arborele este format doar dintr-un singur nod. În general, un cadran trebuie divizat doar dacă el conține pixeli de culori diferite, deci arborele nu va avea neapărat o adâncime uniformă (nu este complet).

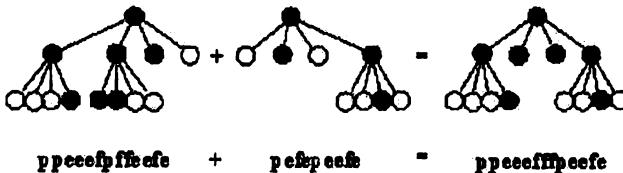
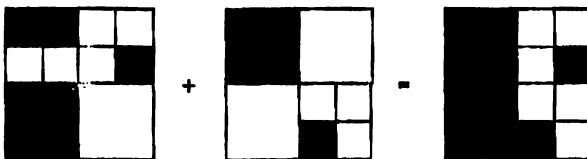
Un fotograf lucrează cu imagini alb-negru pătratice cu 32 de pixeli orizontal și vertical, adică 1024 de pixeli/imaginie. Una dintre operațiile pe care le execută este suprapunerea a două imagini, în scopul formării uneia noi. În imaginea rezultată, un pixel este negru dacă cel puțin un pixel de pe poziția corespunzătoare în una dintre imagini este tot negru, altfel este alb (numai când ambii pixeli sunt albi). Fotograful crede că o fotografie va fi cu atât mai scumpă cu cât are mai mulți pixeli negri. Așa că, înainte de a suprapune două imagini, el trebuie să stie câți pixeli vor fi negri în imaginea rezultată, adică și-ar dorii să stie câți pixeli negri vor rezulta.

Scrieți un program care, pe baza reprezentării ca arbori *Quad* a două fotografii, să calculeze numărul de pixeli negri în fotografie rezultată. În figura de mai jos, este prezentat un exemplu: mai întâi, ordinea împărțirii în cadrane, apoi imaginile, apoi arborii *Quad* de tip *point region*, următe de șirurile în preordine și, în final, de numărul de pixeli negri.

Date de intrare. În fișierul *quad.in* se găsesc mai multe reprezintări de arbori *Quad* de tip *point region*, care sunt consuțerăți în reprezentarea preordonată. Litera *p* simbolizează un nod-tată, *f* un nod-fiu, reprezentând un cadran colorat cu negru, iar *e* desemnează un cadran alb.

Date de ieșire. Pentru fiecare pereche de arbori din fișierul de intrare, scrieți în fișierul *quad.out* trei rânduri, ca fiind:

2	1
3	4



$$448 + 320 - 640$$

quad.in	quad.out
ppeefpfefe pefepefe	ppeefpfefe + pefepefe = ppeeffpfeef 448 + 320 = 640
pe pffe	pe + pffe = pffe 256 + 256 = 512
pceef peepefefe	pceef + peepefefe = peepefefe 256 + 128 = 384
peepefefe	640 + 384 = 832
e pffe	e + pffe = pffe 0 + 768 = 768

(ACM, West European Regional, 1994-1995, zad. 1 mod. 5)

Analiza problemei și proiectarea soluției

Metoda recursivă `unifyTrees()` calculează „suma” (numărul de pixeli negri) a doi arbori `Quad` reprezentați ca giruri în preordine (`t1` și `t2`) și rezultatul este returnat în girul `s`. Atunci când ambii arbori nu sunt frunze (noduri terminale), se vor extrage cei patru subarbori corespunzători cadranelor și metoda este apelată recursiv pentru fiecare dintre cei patru (subarborele `i` al primului arbore este unificat cu subarborele `i` al celui

de-al doilea arbore, pentru $i = 1, \dots, 4$). Pentru a extrage un subarbore, care începe la o poziție dată, folosim metoda:

```
void subTree(string& ss, string& sr, int &t) {...}
```

unde sirul ss este reprezentarea arborelului dat, sr este reprezentarea subarborelui care începe la poziția t în ss . Sirul dat se va parcurge caracter cu caracter de la stânga spre dreapta, începând cu poziția t . În variabila n , vom salva numărul de noduri care mai trebuie să fie citite. Dacă n devine zero, atunci știm că am determinat un subarbore. Atunci când se citește un caracter p , știm că mai trebuie să citim patru noduri succesoare; de aceea, n se incrementează cu 4. Deoarece p nu poate fi un nod-frunză, rezultă că n va fi micșorat cu 1. În total, se vor aduna în acest caz 3. Atunci când se citește o literă f sau e (frunză), atunci n se micșorează cu 1.

Program

```
#include <iostream>
#include <string>

using std::string;
using std::ifstream;
using std::ofstream;

bool isLeaf(char c){
    return
        ('A' <= c && c <= 'Z') ||
        ('a' <= c && c <= 'z');
}

void subTree(string& ss, string& sr, int& t){
    sr.clear();
    if('p'==ss[t]){
        while (n && t<(int)ss.length()){
            if (isLeaf(ss[t])) n--;
            else n += 3;
            sr += ss[t++];
        }
    } else{
        sr=string(1, ss[t]);
        t++;
    }
}

int getValue(string ss, int v){
```

```

if ('p'==ss[0]){
    string sr;
    int t = 0;
    int suma = 0;
    t = 1;
    for(int i=0; i<4; i++){
        subTree(ss, sr, t);
        suma += getValue(sr, v/4);
    }
    return suma;
}
else
    if ('f'== ss[0])
        return v;
    return 0;
}

void unifyTrees(string t1, string t2, string& s){
if ('p'==t1[0] && 'p'==t2[0]) {
    s += "p";
    int i1 = 1, i2 = 1;
    string sr1, sr2;
    for(int i=0; i<4; i++){
        subTree(t1, sr1, i1);
        subTree(t2, sr2, i2);
        unifyTrees(sr1, sr2, s);
    };
}
else
    if (t1[0]=='t' || t2[0]=='f') s += "f";
    else if ('e'==t2[0]) s += t1;
    else if ('e'==t1[0]) s += t2;
}

void write(ofstream& out, string t1, string t2, string s){
out << t1 << " + " << t2 << " = " << s << std::endl;
out << getValue(t1, 1024) << " + "
    << getValue(t2, 1024) << " = "
    << getValue(s, 1024) << std::endl;
out << "-----\n";
}

int main(){
string t1, t2, s;
ifstream in("quad.in");
ofstream out("quad.out");
while(in && !in.eof() && in>>t1>>t2){
    s.clear();
}
}

```

```
    unifyTrees(t1, t2, s);
    write(out, t1, t2, s);
}
return 0;
}
```

Probleme propuse

1. Lungimea laturii unei astfel de fotografii pătratice este n . Calculați aria ce este acoperită de pixeli negri pentru un arbore Quad dat.
 2. Scrieți un program pentru o reprezentare grafică pe cât posibil mai convenabilă a celor doi arbori Quad dați și a arborelui rezultat.
 3. Reduceri de termeni. Un termen este definit inductiv astfel:
 - a) X este termen;
 - b) dacă A și B sunt termeni, atunci $(A \ B)$ este termen;
 - c) un termen se obține doar aplicând regulile de mai sus.

Un termen de forma $((X A) B) C$, unde A , B și C sunt termeni, se poate înlocui cu $((A C)(B C))$. Reducerea unui termen înseamnă aplicarea acestei reguli. Spunem că un termen este redus, dacă nu conține nici un termen reductibil. Un termen se consideră că este subtermen propriu. Scripti un program care să citească termenii și să scrie termeni corespunzători reduși.

Date de intrare. Fișierul *termeni.in* conține mai multe linii cu termeni, câte unul pe linie. Fiecare linie va conține cel mult 80 de caractere. Termenii se consideră corecți sintactic. Liniile de intrare se vor sfârși cu o linie care conține caracterul 0.

Dacă într-o frază există mai multe termeni de lucru se vor scrie în împărțituri, urmând ca prima linie să reprezinte totul cîte o linie reprezentînd termenul respectiv redus, apoi o linie reprezentînd cifre în pozițiile în care termenul de ieșire are paranteze, urmată de o linie vidă. Un termen fără paranteze va fi urmat de două linii vide. Aceste cifre marchează nivelul de imbricare al parantezelor în respectivele pozitii. Acest nivel

ieșire sunt X, (și). Exemplu:

termeni.in	termeni.out
((XX)X)X)	((XX)(XX))
((XX)(XX))X)	31 11 10
(XX)	
0	((XX)((XX)X))
	01 112 2 10
	(XX)
	3 0

Problema 7. Transformata Fourier discretă (DFT)

Transformata Fourier discretă este cel mai important instrument în prelucrarea semnalelor și este folosită pentru codificarea și decodificarea semnalelor la nivelul frecvențelor. Este folosită, de exemplu, de către cunoscutul algoritm de compresie MP3. Scrieți un program care calculează transformata Fourier discretă pentru vectorul de coeficienți $a = (a_0, a_1, \dots, a_{n-1})$.

Fie $\omega_n^0, \omega_n^1, \omega_n^2, \dots, \omega_n^{n-1}$ cele n rădăcini ale ecuației $X^n - 1 = 0$ (rădăcinile unității).

Definim polinomul $P(x) = \sum_{j=0}^{n-1} a_j \cdot x^j$. Transformata Fourier discretă a vectorului a este atunci $y = (P(\omega_n^0), P(\omega_n^1), \dots, P(\omega_n^{n-1}))$ și scriem $y = DFT_n(a)$.

Date de intrare. În fișierul *fourier.in* se găsesc coeficienții vectorului a , ale cărui elemente sunt numere complexe (date prin părțile reală și imaginară). Numărul elementelor lui a este o putere a lui 2.

Date de ieșire. Scrieți în fișierul *fourier.out* transformata Fourier discretă y a vectorului de coeficienți a . Exemplu:

<i>fourier.in</i>	<i>fourier.out</i>
3.45 2.1	17.47 130
0.8 7.6	52.6993 -87.8464
-5.4 3.1	-33.12 10.83
-7.4 2.1	-33.8606 -17.3687
7.6 43.21	11.35 52.82
5.14 32.1	18.8007 32.43
8.76 43	48.5 -12.41
3.12 -1.21	-54.2394 -36.5313

Analiza problemei și proiectarea soluției

Metoda clasică de calculare are complexitatea $O(n^2)$. Dacă în loc de produsele directe, se folosește metoda poate fi îmbunătățită: transformata Fourier rapidă (engl., *Fast Fourier Transformation, FFT*) are complexitatea $O(n \log n)$.

FFT folosește principiul *Divide et Impera*. Metoda utilizează două noi polinoame $A_0(x)$ și $A_1(x)$, undele de grad $n/2$, care să formează dublul lui polinomului $P(x)$, separându-i pe cei corespunzători puterilor pare, respectiv impare:

$$A_0(x) = a_0 + a_2 x + a_4 x^2 + \dots + a_{n-2} x^{\frac{n}{2}-1}$$

$$A_1(x) = a_1 + a_3 x + a_5 x^2 + \dots + a_{n-1} x^{\frac{n}{2}-1}$$

Rezultă: $P(x) = A_0(x^2) + xA_1(x^2)$ (1)

Valorile lui $P(x)$ în $\omega_n^0, \omega_n^1, \omega_n^2, \dots, \omega_n^{n-1}$ vor fi calculate astfel:

1. Calculăm $A_0(x)$ și $A_1(x)$ (ambele de grad $n/2$) pentru valorile $(\omega_n^0)^2, (\omega_n^1)^2, (\omega_n^2)^2, \dots, (\omega_n^{n-1})^2$.
2. Cu ajutorul formulei (1) obținem $P(x)$.

Algoritmul în pseudocod:

```

ALGORITM_FFT(Vector a)
  1.  $n \leftarrow \text{size}(a)$ 
  2. If ( $n = 1$ ) return  $a$ 
  3.  $\omega_n \leftarrow \cos \frac{2\pi}{n} + i \sin \frac{2\pi}{n}$ 
  4.  $\omega \leftarrow 1$ 
  5.  $A_0 \leftarrow (a_0, a_2, \dots, a_{n-2})$ 
  6.  $A_1 \leftarrow (a_1, a_3, \dots, a_{n-1})$ 
  7.  $Y_0 \leftarrow \text{Divide\_Et\_ImperaFFT}(A_0)$ 
  8.  $Y_1 \leftarrow \text{Divide\_Et\_ImperaFFT}(A_1)$ 
  9. For ( $k \leftarrow 0; k \leq n/2 - 1$ ; step 1)
       $y_k \leftarrow Y_0(k) + \omega \cdot Y_1(k)$ 
       $y_{k+\frac{n}{2}} \leftarrow Y_0(k) - \omega \cdot Y_1(k)$ 
       $\omega \leftarrow \omega \cdot \omega_n$ 
    End For
  10. return  $y$ 
END_ALGORITM_FFT(Vector a)
```

Vom scrie clasa *Complex* pentru a reprezenta numerele complexe, unde vom suprăîncărca operatorii $(+, -, *, >>, <<)$. Metoda *FFT()* furnizează transformata Fourier pe baza algoritmului de mai sus. Mai avem nevoie de constantă ω , care poate fi calculată cu una dintre formulele: $\tan \frac{\pi}{4} = 1$ sau $\cos \frac{\pi}{2} = 0 \Rightarrow \pi = 4 \cdot \arctan 1$ sau $\pi = 2 \cdot \arccos 0$.

Program

```
#include <iostream>
#include <vector>
#include <cmath>

double const pi(4*atan(1.0));
using namespace std;
class Complex{
private:
    double _re, _im;
public:
    Complex (){_re=0; _im=0;};
    Complex(double re, double im){
        _re=re; _im=im;
    }
    ~Complex(){};
    Complex operator+(Complex& z);
    Complex operator-(Complex& z);
    Complex operator*(Complex& z);
    friend istream& operator>>(istream&, Complex&);
    friend ostream& operator<<(ostream&, Complex&);
    void setRe(double re){ _re=re;};
    void setIm(double im){ _im=im;};
};

inline Complex Complex::operator+(Complex&z) {
    return Complex(_re+z._re, _im+z._im);
}

inline Complex Complex::operator-(Complex&z) {
    return Complex(_re-z._re, _im-z._im);
}

inline Complex Complex::operator*(Complex&z) {
    return Complex(_re*z._re-_im*z._im, _re*z._im+_im*z._re);
}

istream& operator>>(istream& is, Complex& z) {
    is >>z._re>>z._im;
    return is;
}

ostream& operator<<(ostream& os, Complex& z) {
    os<<z._re<<" "<<z._im;
    return os;
}
```

```

vector<Complex> FFT(vector<Complex> a) {
    int i, n=a.size();
    if(n<=1) return a;
    Complex wn(cos(2*pi/n), sin(2*pi/n));
    Complex w(1.0, 0.0);
    vector<Complex> a0, a1, y0, y1, y;
    for(i=0; i<n; i++){
        if(i%2==0)a0.push_back(a[i]);
        else a1.push_back(a[i]);
        y.push_back(Complex(0, 0));
    }
    y0=FFT(a0);
    y1=FFT(a1);
    for(i=0; i<n/2; i++){
        y[i]=y0[i]+w*y1[i];
        y[i+n/2]=y0[i]-w*y1[i];
        w=w*wn;
    }
    return y;
}

int main(){
    vector<Complex> a, y;
    Complex w, aux, E, res;
    ifstream in("fourier.in");
    ofstream out("fourier.out");
    while(in && !in.eof() && in>>aux){
        a.push_back(aux);
    }
    int n=a.size();
    y=FFT(a);
    for(int i=0; i<y.size(); i++)
        out << y[i] << endl;
}

```

Probleme propuse

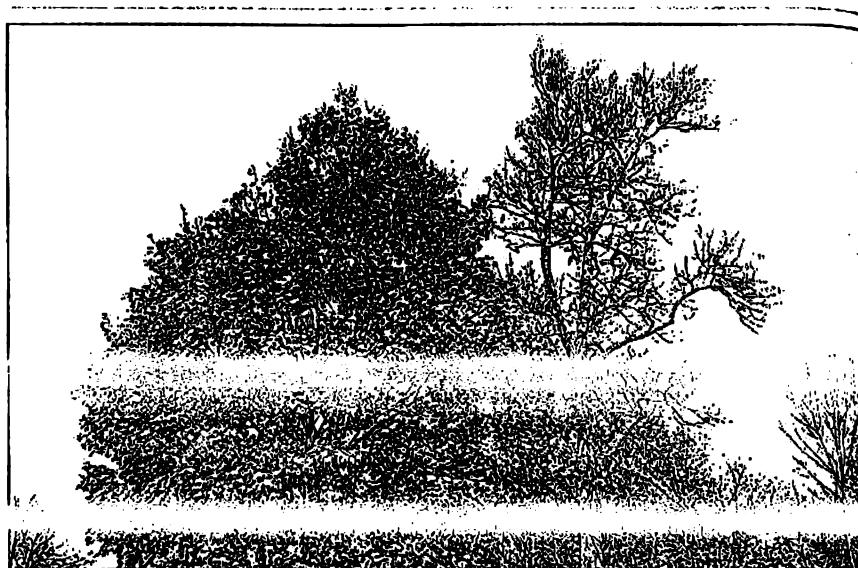
1. Dezvoltăți clasa *Complex* cu operatorii $+$, $*$, $-$ și $/$ și să îl folosiți în metoda *FFT*.
2. Implementați un algoritm naiv de calcul al transformatei Fourier discrete pentru orice grad al polinomului coeficienților și folosiți-o în programul de mai sus. Analizați următoarea variantă:

```

vector<Complex> y;
w.setRe(1);
w.setIm(0);

```

```
for(int k=0; k<n; k++){
    Complex yk(0, 0);
    E.setRe(cos(2*pi*k/n)); E.setIm(sin(2*pi*k/n));
    w.setRe(1); w.setIm(0);
    for(int j=0; j<n; j++){
        yk = yk+a[j]*w;
        w=w*E;
    }
    y.push_back(yk);
}
```



Arbori – toamnă în Ottawa

Capitolul 10

Backtracking

14 probleme complet rezolvate, 64 de probleme propuse

- Problema celor n regine
- Observații generale cu privire la metoda *Backtracking*
- Problema celor n ture pe tabla de șah
- Problema turelor pe primele m linii ale unei tabele de șah
- Problema turelor în scară pe primele m linii ale unei tabele de șah
- Tabăra prieteniei
- Partițiiile unui număr natural
- Referate la geografie
- Tocata drumurilor pe hârtia unei tabele de șah
- Problema fotografiei
- Rostogolirea măngii
- Portocalosport
- Sudoku
- Cășuta lui Moș Crăciun
- Cum să rezolvi o jocură de lăsat
- Încă 12 probleme propuse

*Nu merge unde te duce drumul,
ci mergi unde nu există nici un drum și lasă o urmă.*
Radolph Waldo Emerson

Metoda *Backtracking* este foarte cunoscută și cu ajutorul ei se pot rezolva foarte multe probleme. Una dintre acestea, ce apare în aproape orice curs despre *Backtracking*, este „Problema celor n regine”. Cineva chiar afirma cândva: „Când se spune *Backtracking*, se spune, de fapt, « n regine» și când se spune « n regine» se spune, de fapt, *Backtracking!*”. De aceea, vom începe capitolul cu această problemă, simplu de descris, dar nu chiar aşa de simplu de rezolvat.

Problema 1. Problema celor n regine

Istoric. Problema celor opt regine a fost prezentată pentru prima dată în 1848 de către maestrul bavarez de șah Max Bezzel (1824-1871), în *Berliner Zeitung*, unde acesta se întreba care este numărul de soluții. Acest număr este 92 și a fost introdus de către dr. Franz Nauck în 1850, în ziarul *Leipziger Illustrirten Zeitung*. Pentru că și Carl Friedrich Gauß s-a ocupat de problemă, se indică deseori – fals – că el ar fi răspuns la întrebare. Abia în 1991, B. Bernhardsson prezenta în *ACM SIGART Bulletin*, volumul 2, numărul 7, o soluție pentru problema generalizată. În 1992, Demirörs, Rafraf și Tanik au introdus o echivalență între pătratele magice și problema celor n regine.

Problema. Fie o tablă de șah pătratică de dimensiune $n \times n$ ($4 \leq n \leq 20$). Determinați toate posibilitățile de a ampliaza n regine pe această tablă, astfel încât să nu existe regine care se atacă reciproc. Două regine se atacă reciproc dacă ele se află pe aceeași linie, coloană sau diagonală.

Date de intrare. De la tastatură se introduce numărul de linii n al tablei de șah.

Date de ieșire. Scrieți în fișierul de ieșire *nQueens.out* numărul coloanei pentru fiecare regină ce se află pe linile 1, 2, ..., n respectiv, ca în exemplul de mai jos. Pe ultima linie afișați n și numărul soluțiilor. Exemplu:

Tastatură	<i>nQueens.out</i>
$n = 4$	2 4 1 3 3 1 4 2 4 2
$n = 8$	1 5 8 6 3 7 2 4 1 6 8 3 7 4 2 5 .. 6 3 1 7 5 8 2 4 .. 8 3 1 6 2 5 7 4 8 4 1 3 5 2 7 5 8 92

Analiza problemei și proiectarea soluției

Presupunem că $n = 4$ și că vrem să determinăm toate posibilitățile de a amplasa patru regine pe această tablă, astfel încât oricare două dintre ele să nu se atace reciproc.

Cum putem să procedăm? Mai întâi amplasăm o regină în câmpul $(1, 1)$, după care încercăm să poziționăm o regină pe linia a doua. Observăm că primul câmp posibil este $(2, 3)$, deoarece primele două poziții sunt în conflict cu regina de pe prima linie. Deci a doua regină se amplasează pe poziția $(2, 3)$; vezi figura 10.1.

Apoi, observăm că nu este posibil să poziționăm o regină pe linia a treia: primul câmp este atacat de către regina de pe prima linie, iar următoarele trei câmpuri nu sunt accesibile din cauza reginei de pe a doua linie (figura 10.2). Cu această amplasare a primelor două regine nu are sens să continuăm căutarea unei soluții: este clar că nu există nici una!

Ne întoarcem la a doua linie și deplasăm regina cu c poziție la dreapta (figura 10.3). Pe linia a treia putem așeza o regină pe a doua poziție (figura 10.4). Ultima regină nu poate fi plasată pe al patrulea rând, deoarece se atacă cu reginele deja amplasate (figura 10.5).

Am învățat deja că în acest caz trebuie să ne deplasăm la rândul anterior și să mutăm regina de acolo cu o poziție la dreapta. Remarcăm că nici pe linia a treia nu avem succes, deoarece câmpurile 3 și 4 se atacă cu regina de pe linia a doua. Deci ne deplasăm pe linia a două și observăm că aici ne aflăm pe ultimul câmp.

Trebue aşadar să ne întoarcem la prima linie și să deplasăm regina de aici pe poziția a două (figura 10.6). Pentru a doua regină este valabil numai ultimul câmp (figura 10.7). În oglindă se poate observa de ce pe primul câmp (figura 10.8), fără să se poată păripa, ultima regină poate trecă la treilea câmp (figura 10.9). În acest lucru, ești patru regine sunt amplasate și am determinat deci o soluție a problemei.

Pentru a mai găsi o soluție posibilă, vom urma metoda de la descrisă. După câțiva pași ajungem la soluția din figura 10.10 (se observă că aceasta este o oglindire a

Căutarea a încă unci soluții ne conduce la figura 10.11, caz în care trecerea la rândul anterior (ar fi 0!) nu mai este posibilă. În acest moment, algoritmul se încheie și toate soluțiile pentru $n = 4$ au fost determinate.

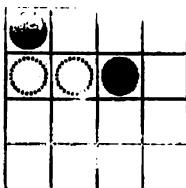


Figura 10.1

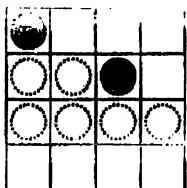


Figura 10.2

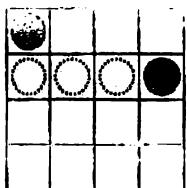


Figura 10.3

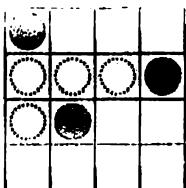


Figura 10.4

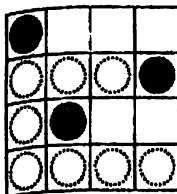


Figura 10.5

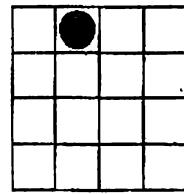


Figura 10.6

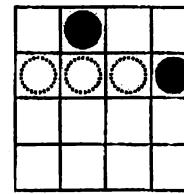


Figura 10.7

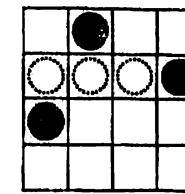


Figura 10.8

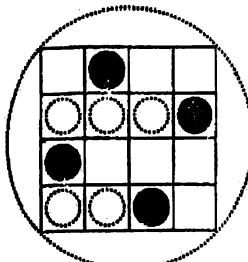


Figura 10.9

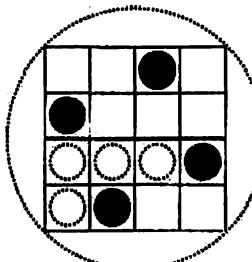


Figura 10.10

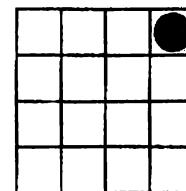
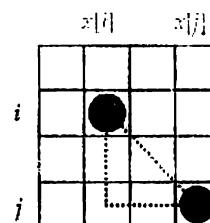
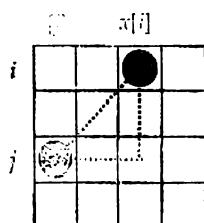


Figura 10.11

Observații:

1. Pe fiecare rând se află exact o regină.
2. Pe fiecare coloană se află exact o regină.
3. Putem reprezenta soluția ca un tablou unidimensional $x[i]$ cu semnificația: $x[i]$ este coloana pe care se plasează a i -a regină. Într-o soluție se poate reprezenta ca $x[1] = [2, 4, 1, 3]$, adică coloana 1, ceea ce înseamnă că regina 1 este plasată în coloana 2, regina 2 în coloana 4, regina 3 în coloana 1 și regina 4 în coloana 3. Această reprezentare ne asigură că pe fiecare rând se află numai o regină.
4. O condiție necesară ca reginele i și j să nu se atace este: $x[i] \neq x[j]$ (coloanele trebuie să fie diferite).
5. Pentru a verifica dacă două regine se atacă sau nu, trebuie să ne asigurăm că ele să se atle pe aceeași diagonală. Putem avea una dintre următoarele posibilități:



- în primul caz, condiția este: $j - i = x[i] - x[j]$;
- pentru al doilea caz, condiția este: $j - i = x[j] - x[i]$.

Considerate împreună, cele două condiții se traduc prin:

$$|j - i| = |x[i] - x[j]|.$$

Algoritmul poate fi astfel prezentat în pseudocod:

ALGORITM_N_REGINE

```

line ← 1
x[1] ← 0
While (line > 0) Execute
    pos ← prima poziție neanalizată de pe linia line, care
        nu se atacă cu reginele de pe pozițiile 1, 2, ..., line – 1
    If (pos există) Then
        x[line] ← pos
        If (line = n) Then
            WriteSolution(x[1], x[2], ..., x[n])
        End_If
        Else
            If (line < n) Then
                line ← line + 1 // următoarea linie
                x[line] ← 0 // setăm coloană de start cu 0
            End_If
            Else Execute
                line ← line – 1 // pas înapoi (backward)
            End_If
        End_If
    End_While

```

END_ALGORITM_N_REGINE

Următorul program implementează acest algoritm. Linile de cod sunt numerotate, pentru a ne putea referi la program în problemele următoare. Notăm programul cu **BP**.

Program BP

```

1: #include <iostream>
2: #include <fstream>
3: #include <vector>

4: using namespace std;

5: void writeSolution(vector<short> &x, ofstream &fout){
6:     for(int i=0; i<x.size(); i++)
7:         fout << x[i]+1 << " ";
8:     fout << endl;
9: }

10: int main(){
11:     short n, k, i, noSol=0;
12:     vector<short> x;
13:     bool flag;
14:     cout << " n = "; cin >> n;
15:     ofstream fout("nRegine.out");
16:     x.push_back(-1);
17:     while(!x.empty()){
18:         k = x.size()-1;
19:         flag = false;
20:         while(!flag && x[k]<n-1){
21:             x[k]++;
22:             flag = true;
23:             for(i=n; i>k; i++)
24:                 if(x[i]==x[k] || abs(x[i]-x[k])==k-i) flag=false;
25:         }
26:         if(flag){
27:             if(k==n-1){
28:                 writeSolution(x, fout);
29:                 noSol++;
30:             }
31:             ...push_back(i);
32:         }else x.pop_back();
33:     }
34:     fout << n << " " << noSol;
35:     return 0;
}

```

Observații. Bucă *while* de pe linia 17 se execută atât timp cât ne aflăm pe un rând valid (între 0 și $n - 1$). Pe linia 20 urmează o buclă *while*, în care se caută următoarea coloană posibilă pentru $x[k]$, astfel încât regina plasată în rândul k , pe coloana $x[k]$ nu se atacă cu reginile predecesoare deja amplasate. Dacă există o poziție „neatacată” $x[k]$ (instrucțiunea *if* de pe linia 25 verifică aceasta), atunci ne aflăm pe ultimul rând (linia 26) – deci am găsit o soluție – sau ne aflăm pe un rând în interior și trebuie să coborâm

pe următorul (linia 30 setează coloana cu 0). Dacă pe rândul k nu există nici un câmp „neacăcat”, atunci ne întoarcem cu un rând înapoi (linia 31).

La rularea programului, remarcăm că timpul de execuție crește foarte repede odată cu n , deoarece complexitatea algoritmului este exponențială.

Observații generale cu privire la metoda Backtracking

Chiar dacă Backtracking îmbunătățește căutarea exhaustivă (de exemplu, nu generăm vectorul $(1\ 1\dots 1)$ și verificăm apoi condițiile ca acesta să fie soluție, ci decidem mult mai devreme că $(1\ \dots)$ nu poate conduce la o soluție!), timpul de execuție crește exponențial. De aceea, metoda Backtracking ajunge să fie folosită doar atunci când s-a verificat că nu există o metodă mai eficientă de rezolvare a problemei (de exemplu, un algoritm polinomial). Pentru clasa problemelor care se rezolvă cu această metodă sunt valabile:

- de obicei, pentru un set de date de intrare se cer toate soluțiile posibile;
- o soluție se poate reprezenta ca un vector, care nu este necesar să aibă lungime fixă;
- o soluție poate fi construită pas cu pas: $x[1], x[2], \dots, x[k]$;
- în momentul k se poate testa dacă valoarea $x[k]$ pentru al k -lea element poate conduce la o soluție, atunci când valorile $x[1], x[2], \dots, x[k-1]$ au fost deja fixate;
- putem să decidem dacă am găsit o soluție a problemei.

În funcție de tipul soluției, o problemă rezolvabilă folosind Backtracking poate fi clasificată astfel: *Backtracking liniar cu lungime fixă* (problema celor n regine), *Backtracking cu lungime variabilă* (generarea permutațiilor unui număr natural), *Backtracking în plan* (toate drumurile calului pe tabla de șah, problema labirintului, problema fotografiei, problema mingii).

În cadrul unei realizări ale algoritmului general pentru Backtracking, presupunând că elementele vectorului $x[]$ pot să ia valori din mulțimile A_1, A_2, \dots, A_n , respectiv A_0 . Algoritmul se poate descrie astfel:

ALGORITM_Backtracking_Iterativ

```

While ( $k \geq 1$ ) Do
    If ( $\exists$ (candidat încă netestat din  $A_k$  pentru  $x[k]$ ) ) Then
         $x[k] \leftarrow$  candidat încă netestat din  $A_k$ 
        pentru al  $k$ -lea element
        If ( $x[1], x[2], \dots, x[k]$  este soluție) Then
            proceseză( $x[1], x[2], \dots, x[k]$ )
        Else  $k \leftarrow k + 1$                                 // pas înainte
        Else  $k \leftarrow k - 1$                                 // pas înapoi
    End_While
END_ALGORITM_Backtracking_Iterativ

```

Metoda *Backtracking* se poate implementa și recursiv, de exemplu, cu un algoritm:

```

ALGORITM_Backtracking_Recurziv(k)
  For (toate valorile posibile pentru  $x[k]$ ) Execute
    If ( $x[1], x[2], \dots, x[k]$  poate conduce la o soluție) Then
      If ( $x[1], x[2], \dots, x[k]$  este soluție) Then
        proceseză  $(x[1], x[2], \dots, x[k])$ 
      Else Backtracking_Recurziv( $k + 1$ )
    End_For
END_ALGORITM_Backtracking_Recurziv(k)
```

Când rezolvăm o problemă folosind *Backtracking recursiv*, executăm apelul *back(1)*.

Probleme propuse

1. Numiți tipurile de soluție pentru problemele rezolvabile folosind metoda *Backtracking*.
2. Modificați programul astfel încât să folosiți tipul `std::stack<short>` și operațiile specifice pentru stivă `top`, `pop` și `push`, în locul vectorului `vector<short>`. Unde apare problema?
3. Scrieți un program ce rezolvă problema recursiv. De exemplu:

```

void back(vector<short> x, short n, ofstream &fout)
{
    short i;
    if (n == 1) {
        cout << x[0] << endl;
        return;
    }
    for (short k = 0; k < n; k++) {
        flag = true;
        if (x.size() == k + 1) x.pop_back();
        for (i = 0; flag && i < k; i++)
            if (x[i] == k + 1) flag = false;
        if (flag) {
            x.push_back(k);
            if (k == n - 1) writeSolution(x, fout);
            else back(x, n, fout);
        }
    }
}
```

Pentru a înțelege mai bine metoda, notați-vă pașii și valorile intermediare pe o foaie de hârtie. De ce, și folosește instrucțiunile `if(x.size() == k - 1) x.pop_back();`?

4. *Pătrat magic*. Un pătrat magic este o matrice pătratică cu numărul de linii și de coloane n , care conține numerele naturale $1, 2, \dots, n^2$ și în care sumele elementelor

de pe aceeași linie, coloană și diagonală este identică. De exemplu, un pătrat magic de dimensiune 4:

$$A = \begin{pmatrix} 16 & 3 & 2 & 13 \\ 5 & 10 & 11 & 8 \\ 9 & 6 & 7 & 12 \\ 4 & 15 & 14 & 1 \end{pmatrix}$$

Demonstrați că pentru un număr natural n dat, suma elementelor de pe fiecare linie, coloană și diagonală este $\frac{n(n^2+1)}{2}$. Scrieți un program care listează în fișierul *patrMagice.out* toate pătratele magice de dimensiune 4. Scrieți și o metodă care să verifice dacă o matrice dată de dimensiune $n \times n$, $4 \leq n \leq 50$ este pătrat magic.

Problema 2. Problema celor n ture pe tabla de șah

Fie o tablă de șah de dimensiune $n \times n$. Determinați toate posibilitățile de a amplasa n ture pe tablă dată, astfel încât să nu existe printre ele perechi care se atacă (două ture se atacă reciproc atunci când se află pe aceeași linie sau pe aceeași coloană).

Date de intrare. De la tastatură, se introduce numărul natural n ($4 \leq n \leq 20$).

Date de ieșire. Scrieți în fișierul *nTure.out* pozițiile coloanelor pentru fiecare tură, ca în exemplul de mai jos, urmate de n și numărul soluțiilor pe ultima linie a fișierului. Exemplu:

	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr><td>1</td><td>2</td><td>3</td><td>4</td></tr> <tr><td>1</td><td>2</td><td>3</td><td>4</td></tr> <tr><td>..</td><td>..</td><td>..</td><td>..</td></tr> <tr><td>5</td><td>4</td><td>3</td><td>1</td></tr> <tr><td>5</td><td>4</td><td>3</td><td>2</td></tr> <tr><td>5</td><td>1</td><td>2</td><td>3</td></tr> </table>	1	2	3	4	1	2	3	4	5	4	3	1	5	4	3	2	5	1	2	3
1	2	3	4																						
1	2	3	4																						
..																						
5	4	3	1																						
5	4	3	2																						
5	1	2	3																						

Analiza problemei și proiectarea soluției

Problema este foarte asemănătoare cu problema reginelor. Condiția este ca oricare două ture să nu se atace reciproc, adică să nu se întâlnească pe aceeași linie sau coloană. Asemănarea ne conduce la lucrul că o soluție a problemei celor n regine este soluție și pentru problema turelor. Vom reprezenta soluția tot cu ajutorul unui vector $x[]$, cu aceeași semnificație. Pentru că dispare o condiție (aceeași diagonală pentru două ture este permisă), numărul soluțiilor crește. Singura schimbare apărută în programul anterior este la pasul în care se testează dacă $x[k]$ este candidat la o soluție parțială, atunci când valorile $x[0], x[1], \dots, x[k-1]$ sunt fixate: nu mai este necesară verificarea diagonalelor. În afara liniei 15 din programul anterior, unde se introduce un alt nume pentru fișierul de ieșire, se va mai modifica și linia 23 astfel:

```
|23: if(x[i]==x[k]) flag=false;
```

(dacă tura i se află pe aceeași coloană cu tura k , înseamnă că vectorul parțial $x[1], x[2], \dots, x[k]$ nu este un candidat pentru o soluție!).

OBSERVAȚIE. Aceasta este, de fapt, generarea tuturor permutărilor, deci numărul soluțiilor pentru un n dat este $n!$.

Probleme propuse

1. Scrieți o implementare recursivă pentru problemă.
2. Permutările sunt listate lexicografic. Modificați programul astfel încât ele să fie listate colexicografic.

Problema 3. Problema turelor pe primele m linii

ale unei table de șah

Găsiți toate posibilitățile de a amplasa un număr maxim de ture pe primele m linii ale unei table de șah pătratice de dimensiune n , astfel încât oricare două dintre ele să nu se atace reciproc ($4 \leq n \leq 20, 1 \leq m \leq n$).

Date de intrare. Valorile n și m se citesc de la tastatură.

Date de ieșire. Toate soluțiile problemei se vor scrie în fișierul *nmTure.out*, iar pe ultima linie se vor scrie n, m și numărul de soluții, ca în exemplu:

	tastatură	nmTure.out
$n = 6$		1 2 3 4
$m = 4$		1 2 3 5
		..
		6 5 4 2
		6 4 ..
		6 4 3 6

Analiza problemei și proiectarea soluției

Numărul maxim de ture cu condițiile date este m . Schimbarea față de probă anterioară este că se vor genera doar primele m elemente ale vectorului. Modificăm programul *BP* cu citirea și scrierea în plus a parametrului m , cu specificarea unui alt fișier de ieșire și cu următoarele două linii:

```
|23: if(x[i]==x[k]) flag=false; //nu se vor testa diagonalele)
```

```
|26: if(k==m-1) { // (o soluție a fost găsită atunci când m elemente sunt
//determinate)
```

Observație:

Accasta este problema generării aranjamentelor de n luate câte m . Numărul soluțiilor este astăzi: $\frac{n!}{(n-m)!} = n(n-1)(n-2)\dots(n-m+1)$.

Probleme propuse

1. Scrieți un program recursiv ce rezolvă problema.
2. Aranjamentele sunt generate lexicografic. Modificați programul astfel încât acestea să fie generate colexicografic.

Problema 4. Problema turelor în scară pe primele m linii ale unei table de șah

Găsiți toate posibilitățile de a amplasa crescător un număr maxim de ture pe primele m linii ale unei table de șah de dimensiune $n \times n$, astfel încât oricare două dintre ele să nu se atace reciproc ($4 \leq n \leq 20, 1 \leq m \leq n$).

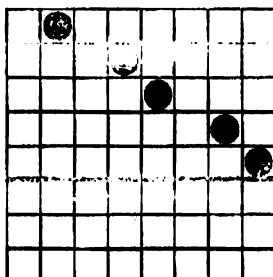
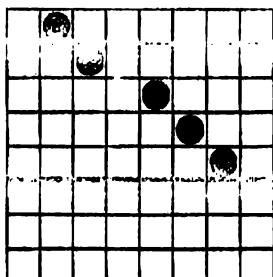
Date de intrare. Numerele naturale n și m se vor introduce de la tastatură.

Date de ieșire. Toate soluțiile se vor scrie în fișierul `nmCTure.out`, pe ultima linie scrieți n , m și numărul soluțiilor, ca în exemplu:

Tastatură	<code>nmCTure.out</code>
$n = 6$	1 2 3 4
$m = 4$	1 2 3 5
	..
	2 4 5 6
	3 4 5 6
	6 4 15

Definiție:

Suntem la scurtura cu tururile $x[1], x[2], \dots, x[m]$ care să verifice că nu există două tururi $x[i], x[j]$, $i < j$, care să verifice că $x[i] = x[j]$ sau $x[i] + j = x[j] + i$. De exemplu, pentru $n = 8$ și $m = 5$ sunt două soluții posibile:



Analiza problemei și proiectarea soluției

Diferența față de problema anterioară este că trebuie să amplasăm crescător cele m elemente. O posibilitate ar fi ca în linia 30 a programului BP să inițializăm $x[k]$ cu $x[k-1]$. Operați în program această modificare.

Observație:

Aceasta este problema combinațiilor de n luate câte m . Numărul soluțiilor este astăzi:

$$\frac{n!}{m!(n-m)!} = \frac{n(n-1)(n-2)\dots(n-m+1)}{m!}$$

Probleme propuse

1. Scrieți o implementare recursivă a programului.
2. Transformați modalitatea lexicografică de listare a combinațiilor în cea colexicografică.

Problema 5. Tabăra prieteniei

În tabăra prieteniei se organizează în fiecare seară un foc de tabără. În prima seară, toți participanții se aşază în jurul focului. În a doua seară aceștia trebuie să se așzeze astfel încât cei care au fost vecini în prima seară să nu mai stea alături. Determinați toate posibilitățile de a-i așeza pe cei n participanți în a doua seară în jurul focului. Presupunem că, în prima seară, aceștia sunt numerotați cu $1, 2, \dots, n$.

Dacă în intrare introducem 3 5 2 4 1, va fi listat numai tabără (dvs. nr. 1).

Date de ieșire. Scrieți toate posibilitățile de așezare în fișierul prietenii.out. Ultima linie va conține n și numărul de posibilități, ca în exemplu:

Tastatură	prietenii.out
	<pre> 3 5 2 4 1 5 3 1 4 2 5 1 0 </pre>

Analiza problemei și proiectarea soluției

În afara condiției că toate numerele din vectorul $x[]$ trebuie să fie diferite două câte două, trebuie să mai verificăm că diferența dintre două elemente vecine nu este egală cu 1 în modul, dar nici cu $n - 1$ (n și 1 au fost vecinul). În programul BP, linile 22 și 23 se vor înlocui cu:

```

if(k==n-1&&(abs(x[k]-x[0])==1||abs(x[k]-x[0])==n-1))
    flag=false;
if(k>0&&(abs(x[k]-x[k-1])==1||abs(x[k]-x[k-1])==n-1))
    flag=false;
for(i=0; i<k; i++) if(x[i]==x[k]) flag=false;

```

Probleme propuse

1. Rezolvați recursiv problema.
2. Găsiți formula de calcul a numărului de soluții în funcție de n .

Problema 6. Partițiile unui număr natural

Fiecare număr natural se poate scrie ca suma altor numere naturale. Pentru toate numerele mai mari decât 1 există mai multe posibilități în acest sens. Scrieți un program care listează toate partițiile unui număr natural ($1 \leq n \leq 75$) și apoi numărul de soluții. Exemplu:

n.in	nPart.out
5	1 + 1 + 1 + 1 + 1 1 + 1 + 1 + 2 1 + 1 + 3 1 + 2 + 2 1 + 4 2 + 3 5 Numar de solutii = 7
10	1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 2 1 + 1 + 1 + 1 + 1 + 1 + 2 + 2 1 + 1 + 1 + 1 + 1 + 3 + 3 1 + 1 + 1 + 1 + 2 + 2 + 2 1 + 1 + 1 + 2 + 2 + 3 1 + 1 + 3 + 3 + 3 1 + 2 + 2 + 2 + 2 + 2 2 + 2 + 2 + 2 + 3 3 + 3 + 3 + 3 50 Numar de solutii = 204226
75	...36 + 39 37 + 38 Numar de solutii = 8118264

Analiza problemei și proiectarea soluției

Care este o soluție a problemei? Iată vectorul $x[1], x[2], \dots, x[k]$ cu proprietățile:

- elementele sale sunt în ordine crescătoare;
- suma lor este n .

Sevența $x[1], x[2], \dots, x[k-1]$ este candidat la o soluție (poate să conducă la construcția unei soluții), dacă elementele sunt în ordine crescătoare și suma lor este mai mică sau egală cu n . Într-o cicleu $x[k]$, initializăm $x[0]$ cu $x[1] - 1$ și folosim un vector $s[i]$, care conține suma primelor i elemente ale vectorului $x[]$. Atunci când $s[i] = n$, înseamnă că am determinat o soluție a problemei.

Program

```

#include <iostream>
#include <vector>

using namespace std;

void writeSolution(vector<int> x, ofstream &out){
    out << x[0];
    for(int i=1; i<x.size(); i++) out << " " + " " << x[i];
    out << endl;
}

int main(){
    unsigned long long noSol=0;
    int n, k;
    vector<int> x, s;
    bool isSuccessor, isCandidate;
    ifstream in("n.in");
    ofstream out("nPart.out");
    in >> n; x.push_back(0);
    while(!x.empty()){
        k=x.size();
        do{
            if(x[k-1]<n){
                isSuccessor=true; x[k-1]++;
                if(s.size()==k) s[k-1]=x[k-1];
                else s.push_back(x[k-1]);
                if(x[k-1]==n-1) isCandidate = s[k-1]<=n;
            }else
                isSuccessor=false;
        }while(isSuccessor && !isCandidate);
        if(isSuccessor){
            writeSolution(x, out);
            noSol++;
        }
        else
            ... = ...; k-1);
    } else{
        x.pop_back();
        if(s.size()==k) s.pop_back();
    }
}
out << "Numar de solutii = " << noSol;
return 0;
}

```

// o soluție cu tablouri

int isSuccessor,

isCandidate;

int s[500];

cout << "n = ";

cin >> n;

k = 1;

x[k] = 0;

s[0]=0;

while(k>0){

do{

if(x[k]<n){

isSuccessor=1;

x[k]++;
 s[k]=s[k-1]+x[k];
 isCandidate=s[k]<=n;

}else
 isSuccessor=0;

}
while((isSuccessor && !isCandidate),
 (s[k]<=n));
if(s[k]==n) Process(x, k)
else{
 ++k;
 x[k]=x[k-1]-1;
}
else
 ...
}

Probleme propuse

1. Scrieți o implementare completă a soluției cu tablouri.
 2. Modificați programul astfel încât partițiile să fie listate în ordine colexicografică.
- Exemplu:

n.in	nPart.out
5	5 2 + 3 1 + 4 1 + 2 + 2 1 + 1 + 3 1 + 1 + 1 + 2 1 + 1 + 1 + 1 + 1 Solution No = 7

3. Modificați programul astfel încât căutarea unui candidat $x[k]$ să se facă cu ajutorul algoritmului din programul BP (problema celor n regine) (în loc de variabilele *isSuccessor* și *isCandidate* folosiți variabila *flag*, iar în locul buclei *do...while* bucla *while()* etc.).
4. Scrieți un program recursiv ce rezolvă problema.
5. Numărul de elemente 1 în toate partițiile este același cu suma numerelor de elemente diferite în fiecare partiție. Demonstrați această propoziție. Exemplu pentru $n = 5$:

partițiile	numărul elementelor diferite
1 1 1 1 1	1
1 1 1 1 0	2
1 + 1 + 3	2
1 + 2 + 2	2
1 + 4	2
2 + 3	2
0	1

Total.

12

Numărul de elemente egale cu 1 în toate partițiile este tot 12. Scrieți un program care verifică propoziția pentru $1 \leq n \leq 40$. Programul trebuie să creeze fișierul *parti.out* de forma:

parti.out
2 2 2 OK
...
5 12 12 OK
...
27 11722 11722 OK
28 14742 14742 OK
...
40 177970 177970 OK

Problema 7. Referate la geografie

Doamna profesoară de geografie a prezentat elevilor o serie de teme pentru referate. Numărul elevilor este n și numărul temelor este m ($m \leq n$), iar fiecare temă trebuie să fie abordată cel puțin o dată. Scrieți un program care să listeze toate posibilitățile de a împărtăși cele m teme de referate celor n elevi.

Date de intrare. Numerele naturale n și m se citesc de la tastatură.

Date de ieșire. Scrieți în fișierul `referat.out` toate modalitățile de împărțire a temelor, ca în exemplu:

Tastatură	referat.out
$n=3$	1 1 2
$m=2$	1 2 1
	1 2 2
	2 1 1
	2 1 2
	2 2 1
	Numar de solutii = 6

Analiza problemei și proiectarea soluției

Definiție:

O funcție $f: A \rightarrow B$ se numește *surjectivă* atunci și numai atunci când $f(A) = B$ (adică $\forall y \in B \exists x \in A : f(x)=y$).

Pentru a rezolva această problemă, trebuie să generăm toate funcțiile surjective cu domeniul $A = \{1, 2, \dots, n\}$ și codomeniul $B = \{1, 2, \dots, m\}$. Aceasta este încă o problemă de tipul *Backtracking liniar cu lungime fixă*. Vom folosi, ca și mai înainte, vectorul x și pe acesta construim soluția cu valoarii posibile și de la $i=1$ până la $i=m$. Într-o buclă *while*, unde se caută o valoare încă netestată pentru $x[i]$, luăm în considerare:

- dacă valoarea pe care a fost deja testată ($x[k] >= 0$...), atunci numărul de valori $x[k]$ în vectorul $vMark[]$ va fi decrementat ($vMark[x[k]]--$);
- în următoarea instrucție *if* vom calcula în *noMarked* numărul de elemente diferite care se află în elementele $x[0], x[1], \dots, x[k]$;
- atunci când suma numărului *noMarked* și a numărului de pozitii rămase ($n-1-k$) este mai mică decât m , atunci variabila *flag* ia valoarea *falsă*; în acest caz, nu putem obține o soluție cu elementele fixate $x[0], x[1], \dots, x[k]$.

Observație

Numărul de funcții surjective cu domeniul $A = \{1, 2, \dots, n\}$ și codomeniu $B = \{1, 2, \dots, m\}$ este:

$$S_j(n, m) = \sum_{i=0}^{m-1} (-1)^i C_m^i (m-i)^n = m^n - C_m^1 \cdot (m-1)^n + \dots + (-1)^{m-1} \cdot m \quad (1)$$

(acest lucru se poate demonstra, de exemplu, prin inducție sau folosind principiul incluziunii și excluziunii din combinatorică). Dacă s-ar cere doar numărul de posibilități pentru împărțirea referelor, atunci putem folosi această formulă.

Program

```
#include <iostream>
#include <fstream>
#include <vector>

using namespace std;

void writeSolution(vector<short> &x, ofstream &fout){
    for(int i=0; i<x.size(); i++)
        fout << x[i]+1 << " ";
    fout << endl;
}

int main(){
    short n, m, k, i, noMarked;
    unsigned long long noSol=0;
    cin >> n >> m;
    bool flag;
    vector<short> x, vMark;
    cout << "n = "; cin >> n;
    cout << "m = "; cin >> m;
    for(i=0; i<n; i++) vMark.push_back(0);
    x.push_back(-1);
    while(!x.empty()){
        k = x.size()-1;
        flag = false;
        while(!flag && x[k]<m){
            if(x[k]>=0) vMark[x[k]]--;
            vMark[++x[k]]++;
            flag = true; noMarked=0;
        }
        if(flag) writeSolution(x, cout);
        else noMarked++;
        if(noMarked==noSol) break;
    }
}
```

```

for(i=0; i<=k; i++)
    if(vMark[i]) noMarked++;
    if(noMarked+n-1-k<m) flag=false;
}
if(flag){
if(k==n-1){
writeSolution(x, out);
noSol++;
} else
    x.push_back(-1);
} else{
    if(x[k]>=0) vMark[x[k]]--;
    x.pop_back();
}
}
out << "Numar de solutii = " << noSol;
return 0;
}

```

Probleme propuse

1. Modificați programul astfel încât soluțiile să fie listate colexicografic.
2. Analizați modalitatea de căutare a elementului candidat $x[k]$ în problema anterioară (folosind variabilele *isCandidate* și *isSuccessor*) și folosiți această căutare în program.
3. Implementați o variantă recursivă.
Demonstrați că este (""). Citeșteți în următoarea problemă de terminație numărul de funcții surjective pe mulțimea $A = \{1, 2, \dots, n\}$ cu valori în mulțimea $B = \{1, 2, \dots, m\}$. Scrieți un program care să determine acest număr. Exemplu:

nm.in	nosurj.out
3 2	
7 4	8400

Problema 8. Toate drumurile calului pe tabla de șah

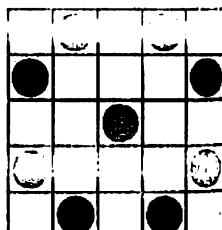
Determinați toate drumurile calului pe o tablă de șah de dimensiune 5×5 atunci când este dat câmpul de start. Toate câmpurile trebuie parcuse o singură dată. Fiecare soluție va fi o matrice pătrată de dimensiuni 5×5 , care simbolizează tabla și conține numerele de la 1 la n^2 în câmpurile corespunzătoare și va fi scrisă în fișierul *cal.out*. Poziția de start va fi notată cu 1, iar câmpurile unde calul se deplasează succesiv vor fi numerotate cu 2, 3, ..., 25, ca în exemplu. Dacă pentru câmpul dat nu există nici o soluție atunci scrieți „nici o soluție” în fișierul de ieșire.



Tastatură	cal.out
4 2	Solutia: 1 19 14 3 8 25 2 9 18 13 4 15 20 5 24 7 10 1 22 17 12 21 16 11 6 23 .. Solutia: 56 23 12 7 4 21 6 17 22 13 8 11 24 5 20 3 16 3 15 5 25 10 19 2 19

Analiza problemei și proiectarea soluției

Acesta este un exemplu clasic pentru *Backtracking în plan*. Dacă un camp nu e ușor, putem să ne treptămuri în cel mult opt poziții. Dacă (x, y) sunt coordonatele unui câmp dat, atunci pozițiile succesoare pot fi: $(x - 2, y - 1)$, $(x - 2, y + 1)$, $(x - 1, y - 2)$, $(x - 1, y + 2)$, $(x + 1, y - 2)$, $(x + 1, y + 2)$, $(x + 2, y - 1)$, $(x + 2, y + 1)$. Vom inițializa toate câmpurile tablei de șah cu -1 și câmpul de start cu 0. Vom scrie o metodă *Backtracking recursivă back()*, ai cărei trei parametri sunt coordonatele câmpului de start și numărul de câmpuri deja parcuse. Decidem că un câmp (x_{new}, y_{new}) este candidat pentru o soluție atunci când câmpul respectiv se află pe tablă și dacă valoarea sa este -1 (acest câmp nu a fost încă parcurs). Atunci când un câmp este candidat la o soluție, vom apela metoda *back()*: *back(*linew*, *cnew*, *step* + 1)*.



Program

```
#include <iostream>
#include <cmath>
#include <fstream>

using namespace std;

int n;
int Table[21][21];
int nSol = 0;
ofstream f("cal.out");

writeSolution(){
    f << " Solution: " << ++nSol << endl;
    for(int i=0; i<n; i++){
        for(int j=0; j<n; j++){
            f.width(3);
            f << Table[i][j] + 1 << " ";
        }
        f << endl;
    }
    return 0;
}

void back(int l, int c, int step){
    if(l==n-1){ //step == n-1
        writeSolution();
        return;
    }
    int dx, dy, lnew, cnew;
    for(dx = -2; dx<3; dx++)
        for(dy = -2; dy<3; dy++)
            if(abs(dx*dy)==2){
                lnew = l + dx;
                cnew = c + dy;
                if(0<=lnew && lnew< n &&
                   0<=cnew && cnew< n &&
                   Table[lnew][cnew]==-1){
                    Table[lnew][cnew] = step+1;
                    back(lnew, cnew, step+1);
                    Table[lnew][cnew] = -1;
                }
            }
}

int main(){
    int l, c;
    n=5;
    for(l=0; l<n; l++)
        for(c=0; c<n; c++)
            Table[l][c] = -1;
}
```

```

for(c=0; c<n; c++)
    Table[l][c] = -1;
cin >> l >> c;
Table[l-1][c-1] = 0;
back(l-1, c-1, 0);
if(0==nSol) f << "nici o solutie";
return 0;
}

```

Probleme propuse

1. Modificați programul astfel încât să nu mai fie folosite variabile globale.
2. Scrieți o implementare recursivă a algoritmului.
3. Ne imaginăm că unele câmpuri ale tablei sunt închise, adică nu este permisă parcurgerea acestora. Modificați problema și programul astfel încât să se determine toate drumurile calului, iar câmpurile închise să nu fie parcuse.

Problema 9. Problema fotografiei

O fotografie alb-negru este dată sub forma unei matrice binare (ce conține doar valoarea 0 și 1). Ea reprezintă mai multe obiecte. Zonele care aparțin aceluiași obiect conțin numai elemente de 1, fundalul conține zerouri. Se cere numărul de obiecte nOb și o nouă reprezentare pentru fotografie, astfel încât obiectele diferite să pe aceasta să fie numerotate cu culori diferite: 2, 3, ..., $nOb + 1$. Doi de 1 care sunt vecini pe linie sau pe coloană trebuie să aparțină aceluiași obiect.

Date de intrare. În fișierul foto.in, se află pe prima linie dimensiunile matricii (a fotografiei), care nu depășesc 100, urmate de fotografia alb-negru în forma unei matrice binare.

Date de ieșire. Scrieți în fișierul foto.out pe prima linie numărul de obiecte apoi fotografia colorată, astfel încât obiectele să aibă culorile 2, 3, ..., $nOb + 1$, ca în exemplu:

foto.in	foto.out
4 4	3
1 1 0 1 1	2 2 0 3 3
0 0 0 1 0	0 0 0 3 0
1 1 0 1 1	4 4 0 3 3
1 0 1 1 0	4 0 3 3 0

Analiza problemei și proiectarea soluției

Vom scrie metoda recursivă `color()`, care verifică dacă punctul dat ca parametru se află în interiorul fotografiei și are valoarea 1. În acest caz, colorăm toți vecinii punctului cu valoarea curentă și apelăm metoda pentru fiecare vecin.

Program

```
#include <iostream>

using namespace std;
int a[30][30], m, n;

color(int i, int j, int cul){
    if(0<=i && i<m && 0<=j && j<n && 1==a[i][j]){
        a[i][j]=cul;
        color(i, j-1, cul);
        color(i, j+1, cul);
        color(i-1, j, cul);
        color(i+1, j, cul);
    }
}

int main(){
    int i, j, cul;
    ifstream fin("foto.in");
    ofstream fout ("foto.out");
    fin >> m >> n;
    for(i=0; i<m; i++)
        for(j=0; j<n; j++)
            fin >> a[i][j];
    cul=1;
    for(i=0; i<m; i++){
        for(j=0; j<n; j++)
            if(1==a[i][j]) color(i, j, ++cul);
        fout << --cul;
    }
    for(i=0; i<m; i++){
        fout << cul;
        for(j=0; j<n; j++)
            fout << a[i][j] << " ";
    }
    return 0;
}
```

Probleme propuse

1. Modificați programul astfel încât să nu mai fie folosite variabilele globale.

2. Presupunem că și două puncte care sunt vecine pe diagonală aparțin aceluiași obiect; cu această nouă condiție în plus, un punct va putea avea maximum opt vecini. Țineți cont de aceasta în program.

Problema 10. Rostogolirea mingii

Considerăm o matrice de dimensiuni m și n , ale cărei elemente sunt înălțimile unităților (metri) într-un câmp de joc. Undeva în acest câmp se plasează o minge, care poate să alunece doar pe un câmp vecin cu o înălțime mai mică decât cel pe care se află. Un vecin se află într-unul dintre cele patru puncte cardinale. Găsiți toate posibilitățile prin care mingea se poate deplasa în acest fel în afara câmpului (ajunge pe o linie sau pe o coloană de margine a matricei). Presupunem că valorile înălțimilor sunt alese în aşa fel, încât întotdeauna există cel puțin o soluție.

Date de intrare. În fișierul *minge.in*, se găsesc pe prima linie valorile m și n ($4 \leq m, n \leq 20$), pe următoarele m linii se află înălțimile câmpului, iar pe ultima linie se află coordonatele câmpului de start.

Date de ieșire. Se vor scrie în fișierul *minge.out*. Fiecare linie a fișierului de ieșire conține câte un drum posibil al mingii către marginea câmpului, ca în exemplu:

<i>minge.in</i>	<i>minge.out</i>
4 4	(3, 2) (2, 2) (1, 2)
0 1	(3, 2) (2, 2) (2, 1) (1, 2)
3 2 1 3	(1, 2) (3, 1)
4 5 2 3	(3, 2) (3, 3) (2, 3) (1, 3)
3 6 1 0	(3, 2) (3, 3) (4, 3)
3 2	



Soluția utilizează *Backtracking în plan*. Vom codifica fiecare poziție cu un număr natural: câmpului (i, j) , cu $0 \leq i < m$, $0 \leq j < n$, îi asociem valoarea $k \leftarrow i \cdot n + j$. Pentru astfel de valoare k , se vor determina linia și coloana corespunzătoare prin: $i \leftarrow k \text{ div } n$, $j \leftarrow k \text{ mod } n$. Ele sunt și noile conditii de vecinătate pentru că într-o matrice de natură să loc de perechi (linie, coloană). Implementăm metodele ajutătoare *onTheBorder()* (testeză dacă un câmp al matricei se află pe marginea acesteia) și *onTheTable()* (testeză dacă o pereche de numere reprezintă un câmp de pe matrice). Metoda care găsește toate drumurile este metoda recursivă *back()*. Condiția de ieșire din metodă este indeplinită atunci cand ultimul element al vectorului $x[]$ se atâră pe conturul matricei. Atunci când nu ne aflăm pe acest contur, va fi apelată recursiv metoda pentru toți vecinii câmpului curent care au o înălțime mai mică decât el.

program

```

#include <fstream>
#include <vector>

using namespace std;

bool onTheBorder(int k, short m, short n) {
    short l = k/n;
    short c = k%n;
    bool flag=false;
    if(0==l || m-1==l) flag=true;
    if(0==c || n-1==c) flag=true;
    return flag;
}

bool onTheTable(short line, short col, short m, short n) {
    return
        (0<=line && line<m) &&
        (0<=col && col<n);
}

void writeSolution(vector<int> &x, short T[][100],
                   short n, ofstream &out){
    for(int i=0; i<x.size(); i++){
        out << "(" << x[i]/n+1 << ", " << x[i]%n+1 << ") ";
    }
    out << endl;
}

void back(vector<int> x, short T[][100], short m,
          short n, ofstream &out){
    if(onTheBorder(x[k], m, n)){
        writeSolution(x, T, n, out);
        return;
    }
    short l = x[k]/n;
    short c = x[k]%n;
    short lnew, cnew, dx, dy;
    for(dx=-1; dx<2; dx++)
        for(dy=-1; dy<2; dy++)
            if(l==abs(dx+dy)){
                lnew = l+dx;
                cnew = c+dy;
                if(onTheTable(lnew, cnew, m, n)&&T[lnew][cnew]<T[l][c]){
                    x.push_back(lnew*n+cnew);
                }
            }
}

```

for(dx=-1; dx<2; dx++)
 for(dy=-1; dy<2; dy++)
 if(l==abs(dx+dy))...

Furnizează pentru (dx, dy) valorile $(-1, 0), (1, 0), (0, -1), (0, 1)$, adică direcțiile N, S, V, E. Câmpurile vecine sunt astăzi (l_{new}, c_{new}) : $(l-1, c), (l+1, c), (l, c-1), (l, c+1)$. Când nouă element se află pe matrice și înălțimea să este mai mică decât cea actuală, atunci facem un pas mai departe...

```

        back(x, T, m, n, out);
        x.pop_back();
    }
}

int main(){
    ifstream in("minge.in");
    ofstream out("minge.out");
    short i, j, m, n;
    short l0, c0;
    short T[100][100];
    vector<int> x;
    in >> m >> n;
    for(i=0; i<m; i++)
        for(j=0; j<n; j++)
            in >> T[i][j];
    in >> l0 >> c0;
    l0--; c0--;
    x.push_back(l0*n+c0);
    back(x, T, m, n, out);
    return 0;
}

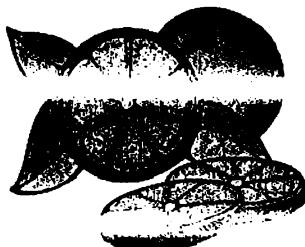
```

Probleme propuse

1. Dezvoltați mai departe programul astfel încât să fie afișat și numărul soluțiilor și mijloacele de a obține rezultatul.
2. Scrieți un program iterativ pentru problema.

Problema 11. Portocalosport

Jocul Portocalosport pare să fie simplu, dar vom vedea că poate deveni problematic. Iată mai întâi vom numerota 100 de portocale cu o cerneală netoxică de la 1 la 100. Apoi toate vor fi aşezate într-un platou uriaș pe mijlocul mesei. Cei doi jucători trebuie să mânânce jocul de la intr-un sfârșit la altă de timp (să zicem 15 minute) și să calculeze produsul numerelor scrise pe cele mâncate. Portocala cu numărul 1 nu are voie să fie mâncată împreună cu alte portocale; aceasta înseamnă că unul dintre jucători are voie să o mânânce doar la început și după aceea să nu mai consume altele. Atunci când un jucător raportează la sfârșit numărul 1, atunci a mâncat doar portocala 1, când numărul este mai mare decât 1, atunci nu a mâncat-o sigur. Atunci când timpul de mâncat a expirat, cei doi jucători



trebuie să raporteze produsele numerelor de pe portocalele mâncate (scorurile). Scorurile raportate nu au voie să fie identice (dacă e aşa, atunci jucătorii vor mai fi însăşi să mânânce câte o portocală din platou). În mod normal, va câştiga jocul jucătorul care raportează un scor mai mare. Spunem „în mod normal”, deoarece unele dintre jucători ar fi putut să mintă sau să calculeze greşit produsul. Este simplu să se imagină că se ajunge la controverse, care trebuie mai întâi rezolvate, înainte de a se declara câştigătorul final. Atunci când anumite condiţii sunt satisfăcute, este posibil să câştige şi jucătorul care a declarat un scor mai mic. Presupunem că jucătorul cu scor mai mic spune întotdeauna adevarul atunci când scorul raportat de el se poate descompune ca produs de numere diferite din mulțimea {1, 2, ..., 100}. În acest caz, el poate să câştige atunci când se demonstrează că jucătorul celălalt a mintit. Această înuinătătură poate fi dovedită prin faptul că produsul său se poate descompune în mulțimea de pe portocale care au fost mâncate și de celălalt jucător. Jucătorul cu scor mai mare poate să câştige atunci când nu se poate demonstra că a mintit, adică există o modalitate de a mâncă portocalele care să nu se contrazică cu a celuilalt. Când nici unul dintre rezultate nu se poate descompune cu ajutorul factorilor din {1, 2, ..., 100}, atunci este remiză.

Exemplul 1. Când primul jucător spune 343, iar al doilea 49, atunci primul jucător minte. Singura modalitate de a obține 343 este să fi mâncat portocalele 7 și 49. Singura modalitate de a obține 49 este să fi mâncat portocala 49. De aici, deducem că primul jucător minte (al doilea spune adevarul prin definiție!).

Exemplul 2. Când primul jucător spune 194 și al doilea 178, atunci singura modalitate este acela că primul jucător să fi mâncat portocalele 2 și 97. Al doilea spune adevarul, deci posibilitatea 2 și 89. Întrucătă că al doilea spune adevarul prin definire, rezultă că primul minte, deci al doilea câştigă (nu se putea să mânânce amândoi portocala 2!).

Exemplul 3. Scoruri declarate: 138 și 258. Pentru a obține 138, avem posibilitățile (6, 23), (3, 46), (2, 69) și (2, 3, 23). Putem presupune că primul a mâncat portocalele 6 și 23, iar al doilea spune adevarul.

Exemplul 4. Scoruri declarate: 1236 și 100. Nu există nici o posibilitate de a obține 1236, deci primul jucător minte. Pentru a obține 100, avem posibilitățile (100), (5, 20), (4, 25), (2, 50) și (2, 5, 10), deci al doilea spune adevarul și câştigă jocul.

Din păcate, toți care au luat parte la joc au mâncat atât de multe portocale, încât nu mai sunt în stare să facă astfel de calcule complicate. Scrieți un program care să rezolve problema automat.

Date de intrare. În fișierul *porto.in* se găsesc una sau mai multe perechi de scoruri declarate, côte una pe linie. Presupunem că într-o secvență de numere de tipul *unsigned long long*.

Date de ieșire. Pentru fiecare pereche din fișierul de intrare scrieți rezultatul pe o linie din fișierului *porto.out*, care poate fi una dintre propozițiile:

Castiga primul jucator!
 Castiga al doilea jucator!
 Primul s-a incurcat! Castiga al doilea!
 Al doilea s-a incurcat! Castiga primul!
 Amandozi s-au incurcat! Remiza!

Primele două propoziții vor fi scrise atunci când ambele rezultate se pot descompune în factori diferenți între 1 și 100, iar jucătorul cu scor mai mare câștigă atunci când nu a mințit. A treia și a patra propoziție se vor scrie atunci când scorul declarat de exact unul dintre jucători nu se poate descompune ca produs de numere diferențe dintre 1 și 100. Ultima propoziție se va afișa atunci când ambele scoruri declarate nu pot fi obținute conform regulilor. Exemplu:

porto.in	porto.out
110 119	Castiga al doilea jucator!
294 202	Al doileas s-a incurcat! Castiga primul!
343 49	Castiga al doilea jucator!
610 3599	Castiga primul jucator!
138 258	Castiga al doilea jucator!
941 2234	Amandozi s-au incurcat! Remiza!
1236 100	Primul s-a incurcat! Castiga al doilea!
6231 1500	Castiga primul jucator!
151 127	Amandozi s-au incurcat! Remiza!
1 101	110 doileas s-a incurcat! Castiga primul!

(inspirat de ACFI South Central USA Collegiate Programming Contest, 2000, varianta)

Analiza problemei și proiectarea soluției

Problema se reduce la determinarea tuturor posibilităților de a descompune un număr natural n dat, toate descompunerile ca produs de numere diferențe între 1 și 100, și să se verifice dacă rezultatul este corect sau nu. Aceste descompuneri pentru ambiții jucători sunt identificate, le comparăm două câte două pentru a decide dacă există o pereche de descompuneri disjuncte.

Pentru un număr natural n dat, toate descompunerile ca produs de numere diferențe între 1 și 100 sunt generale între numerele de mulțime (de tipul $1 \dots 100$). Metoda *divVect(UINT n, vector<short>& a)* furnizează în vectorul *a* toți divizorii lui *n* din mulțimea $\{2, \dots, 100\}$. Acest vector *a* va fi folosit de metoda *Backtracking* recursivă *back()*. Această metodă construiește vectorul *x* ca fiind vectorul caracteristic și unui submulțimii din *n* ($x[k] = 1$, dacă elementul *k* din *n* este introdus în soluție; $x[k] = 0$, dacă elementul *k* din *n* nu este introdus în soluție). Iată cum funcționează astfel:

- produsul elementelor din soluția parțială curentă se salvează în *p*;

dacă $p = n$, atunci am determinat o descompunere pentru n , care va fi introdusă în vectorul cu mulțimi cu ajutorul metodei `addSet()`;
 dacă $p \neq n$, atunci setăm valoarea $x[k]$ mai întâi *false* și apoi *true*;
 $x[k]$ ia valoare *false*, aceasta înseamnă că divizorul k este introdus în soluție, apoi apelăm metoda `back()` pentru următorul nivel ($k + 1$);
 $x[k]$ ia valoare *true*, al k -lea divizor ($a[k-1]$ în program) poate să fie adăugat soluției parțiale numai dacă $(p \cdot a[k-1])$ este un divizor al lui n (poate conduce la obținerea unei descompuneri valide). O altă condiție este ca $\frac{n}{p \cdot a[k-1]} = 1$ ($a[k-1]$ va fi ultimul divizor în descompunere) sau $\frac{n}{p \cdot a[k-1]} > a[k-1]$ (elementele descompunerii sunt considerate în ordine crescătoare):

```
if((n/p)%a[k-1]==0){                                //n divizibil cu p*a[k-1]
    x[k]=1;
    if((n/p)/a[k-1]==1 || (n/p)/a[k-1]>a[k-1])  {...}...
```

Atunci când o descompunere validă a fost găsită, aceasta va fi adăugată vectorului cu soluții, cu ajutorul metodelor:

```
void addSet(int k, TSetsVector& b) {...}
```

Vectorul b complet cu soluții corespunzătoare parametrului n se va construi cu

```
void buildSetsVector(ULLINT n, TSetsVector& b){
    divVect(n, a);
    b.clear(); x.reset();
    back(n, 1, b);
```

Pentru punctajele citite, $n1$ și $n2$, se vor construi vectorii cu descompunerile b și c în programul principal cu ajutorul:

```
buildSetsVector(n1, b);
buildSetsVector(n2, c);
```

Metoda `makeDecision()` stabilește care este rezultatul jocului pe baza vectorilor de descompuneri b și c . Ea folosește funcția `isADisjointCombination(TSetsVector b, TSetsVector c)`, care decide dacă există o descompunere în b și o descompunere în c care sunt disjuncte (mulțimile corespunzătoare au intersecția vidă).

Remarcăm utilizarea tipului `bitset<N>` pentru stocarea vectorului caracteristic x în metoda recursivă *Backtracking*. Un astfel de element se diferențiază de tipul `vector<bool>` prin faptul că are lungime fixă.

Program

```
#include <iostream>
#include <set>
#include <vector>
#include <bitset>

using namespace std;

typedef unsigned long long ULLINT;
typedef set<short> TSet;
typedef vector<TSet> TSetsVector;

vector<short> a;
bitset<30> x;

void addSet(int k, TSetsVector& b) {
    int i, t=1;
    TSet auxSet;
    for(i=1; i<k; i++)
        if(x[i])
            auxSet.insert(a[i-1]);
    b.push_back(auxSet);
}

void back(ULLINT n, int k, TSetsVector& b){
    int i, j;
    long int p = 1;
    int aSiz = (int)a.size();
    if(p == n) {
        addSet(k, b);
    }
    else
        for(i=0; i<2; i++){
            x[k] = 0==i ? false:true;
            if(!i) back(n, k+1, b);
            else
                if(0==(n/p)&a[k-1]){
                    x[k]=true;
                    if(i==(n/p)/a[k-1] || (n/p)/a[k-1]>a[k-1])
                        back(n, k+1, b);
                }
        }
}
```

```
}

void divVect(ULLINT n, vector<short>& a){
    a.clear();
    for(int i=2; i<=100; i++)
        if(0==n% i)
            a.push_back(i);

void buildSetsVector(ULLINT n, TSetsVector& b){
    divVect(n, a);
    b.clear(); x.reset();
    back(n, 1, b);

    bool isACommonElement(TSet A, TSet B){
        bool flag = false;
        set<short>::const_iterator itA;
        for (itA = A.begin(); !flag && itA != A.end(); itA++)
            if (B.find(*itA) != B.end()){
                flag = true;
            }
        return flag;
    }

    bool isADisjointCombination(TSetsVector b, TSetsVector c){
        bool flag = false;
        short bSiz = (short) b.size(), cSiz = (short)c.size();
        for(short i=0; !flag && i<bSiz; i++)
            for(short j=0; !flag && j<cSiz; j++)
                if(!isACommonElement(b[i], c[j])) flag=true;
        return flag;
    }

    void makeDecision(ULLINT n1, ULLINT n2,
                      TSetsVector b, TSetsVector c,
                      ofstream &fout){

        if(!b.size() && !c.size())
            fout << "Amandoi s-au incurcat! Remiza!" << endl;
        else if(!b.size())
            fout << "Primul s-a incurcat! Castiga al doilea!" << endl;
        else if(!c.size())
            fout << "Al doilea s-a incurcat! Castiga primul!" << endl;
        else
            if(n1<n2)
```

```

        if(isADisjointCombination(b, c))
            fout << "Castiga al doilea jucator!" << endl;
        else
            fout << "Castiga primul jucator!" << endl;
    else
        if (isADisjointCombination(b, c))
            fout << "Castiga primul jucator!" << endl;
        else
            fout << "Castiga al doilea jucator!" << endl;
    }

int main(){
    ULLINT n1, n2;
    TSetsVector b, c;
    ifstream fin("porto.in");
    ofstream fout("porto.out");
    while(fin && !fin.eof() && fin>>n1>>n2){
        buildSetsVector(n1, b);
        buildSetsVector(n2, c);
        makeDecision(n1, n2, b, c, fout);
    };
    fin.close(); fout.close();
    return 0;
}

```

Probleme propuse

- Modificați programul astfel încât să nu mai fie utilizate variabilele globale.
Vorbiți cădun jucător și să le furnizați și o posibilitate de a scrie în fișiere rezultatele (când este posibil, căte una pentru fiecare jucător).
- Scrieți un program C care să rezolve problema (de exemplu, stocați toate descompunerile într-o matrice bidimensională).
Puteți să folosiți clasele din `<map>` și `<set>` din STL: descriere, metode, operatori și exemple.
- Containerele asociative sunt cele în care informațiile sunt stocate ordonat. Citiiți în *Help* despre containerele asociative `std::set` (conceptul de *multime* din matematică, în care fiecare element este luat o singură dată) și `std::multiset` (unde elementele pot să se repete).
- În *header-ul* `<algorithm>` din *STL* se găsesc și metode corespunzătoare operațiilor cu mulțimi: *includes* (decide dacă o mulțime *A* este inclusă în altă mulțime *B*), *set_union* (furnizează reuniunea a două mulțimi), *set_intersection* (intersecția) și *set_symmetric_difference* (diferența simetrică a două mulțimi *A* și *B*, c. z. $A \Delta B = (A - B) \cup (B - A)$). Modificați programul de mai sus astfel încât în locul metodei *isACommonElement(TSet A, TSet B)* să folosiți metoda *set_intersection*. Scrieți

programe demonstrative implementând toate cele patru metode pentru operațiile pe mulțimi. Scrieți funcții echivalente pentru ele (presupunând că acestea nu există). Scrieți o metodă *Backtracking* iterativă în locul celei recursive.

Implementați o metodă care furnizează pentru un număr natural (*unsigned long long*) toate descompunerile ca produs de factori diferenți din mulțimea {1, 2, ..., 100}.

Problema 12. Sudoku

Cine nu a auzit de popularul joc de gândire Sudoku? Predecesoare timpurii ale jocului pot fi considerate, de exemplu, *pătratele latine* ale celebrului matematician Leonhard Euler (1707-1783), care a prezentat o serie încă în secolul al XVIII-lea. El le numea *carre latin* și nu erau împărțite și în subpătrate, ca în cunoșutele careuri Sudoku de astăzi. Jocul actual se compune dintr-un careu pătratic cu nouă linii și nouă coloane, care se poate împărți și în 3×3 blocuri, fiecare cu dimensiunea de 3×3 câmpuri. În unele câmpuri se găsesc deja cifre de la 1 la 9; în mod normal, sunt deja completeate între 22 și 36 de câmpuri din cele 81. Scopul este acela de a fi completeate și câmpurile libere cu cifre de la 1 la 9, astfel încât pe toate liniile, coloanele și blocurile fiecare cifră să apară doar o singură dată. Atunci când un număr este posibil într-un anumit câmp, acesta este numit „candidat”. Cele trei zone (linie, coloană, bloc) sunt numite „unități”. Chiar dacă de obicei Sudoku conține cifre, nu sunt necesare nici un fel de calcule, în locul acestora putând fi folosite orice simboluri. Exemplu:

			5		8	9	3	6
				3		5	4	
9	3	5	4			7	8	
	7	9		8			5	
1			7	4			2	
7			2	5				3
4					7	2		

1	4	7	5	2	8	9	3	6
8	6	3	2		3		5	4
9	3	5	4				7	8
6	7	9	9	8	2	4	1	5
3	1		7	4			2	
7			2	5				3
4	5	6	8	3	9	7	2	1
2	8	3	1		5	6	9	7

Exemplu de puzzle Sudoku

Problema cere rezolvarea unui Sudoku care se află în fișierul *sudoku.in*, în care câmpurile libere sunt marcate cu „*”. Rezultatul se va scrie în *sudoku.out*, unde se vor lista toate posibilitățile. Dacă nu există soluție, se va scrie *nu există soluție*. Exemplu:

<i>sudoku.in</i>	<i>sudoku.out</i>
***5*8936	1 4 7 5 2 8 9 3 6
****3*54	8 6 2 7 9 3 1 5 4
9354****78	9 3 5 4 6 1 2 7 8
679*8***5	6 7 9 3 8 2 4 1 5
*****8*	3 2 4 9 1 5 6 8 7
*1**74***2	5 1 8 6 7 4 3 9 2
7**25***3	7 9 1 2 5 6 8 4 3
4*****72*	4 5 6 8 3 9 7 2 1
1**	2 8 3 1 4 7 5 6 9
*1***94**	8 1 5 7 3 9 4 2 6
*74***39*	6 7 4 2 8 5 3 9 1
3*****5**	3 9 2 4 1 6 5 8 7
2***5*673	2 8 1 9 5 4 6 7 3
7***63*4*	7 5 9 8 6 3 1 4 2
628**	4 3 6 1 7 2 8 5 9
*****	9 2 3 5 4 1 7 6 8
7*2*35	1 4 7 6 2 8 9 3 5
568*9**1*	5 6 8 3 9 7 2 1 4

Analiza problemei și proiectarea soluției

Dacă ne referim la primul exemplu prezentat și numerotăm liniile și coloanele de la 0 la 8, vom avea următoarea reprezentare a pozitiei acelor numere: (0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7), (0, 8), (1, 0), (1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (1, 7), (1, 8), (2, 0), (2, 1), (2, 2), (2, 3), (2, 4), (2, 5), (2, 6), (2, 7), (2, 8), (3, 0), (3, 1), (3, 2), (3, 3), (3, 4), (3, 5), (3, 6), (3, 7), (3, 8), (4, 0), (4, 1), (4, 2), (4, 3), (4, 4), (4, 5), (4, 6), (4, 7), (4, 8), (5, 0), (5, 1), (5, 2), (5, 3), (5, 4), (5, 5), (5, 6), (5, 7), (5, 8), (6, 0), (6, 1), (6, 2), (6, 3), (6, 4), (6, 5), (6, 6), (6, 7), (6, 8), (7, 0), (7, 1), (7, 2), (7, 3), (7, 4), (7, 5), (7, 6), (7, 7), (7, 8), (8, 0), (8, 1), (8, 2), (8, 3), (8, 4), (8, 5), (8, 6), (8, 7), (8, 8). Ele vor fi salvate în program în vectorul *poz* cu perechi de tip *TPair*, fiecare pereche reprezentând linia și coloana unui camp liber.

Vom construi mulțimi cu unitățile pe care trebuie să se afle de fiecare dată toată o coloană sau o linie: cele 9 cifre: liniile, coloane și blocuri 3×3 :

Linia 0: $sl[0] = \{3, 6, 7, 8, 9\}$

Coloana 1: $sc[0] = \{4, 6, 7, 9\}$

Linia 2: $sl[1] = \{3, 4, 5\}$

Coloana 2: $sc[1] = \{1, 3, 7\}$

Linia 3: $sl[2] = \{3, 4, 5, 7, 8, 9\}$

Coloana 3: $sc[3] = \{5, 9\}$

Linia 4: $sl[3] = \{5, 6, 7, 8, 9\}$

Coloana 4: $sc[3] = \{1, 2, 4, 5\}$

Linia 5: $sl[4] = \{8\}$

Coloana 5: $sc[4] = \{5, 7, 8\}$

Linia 6: $sl[5] = \{1, 2, 4, 7\}$

Coloana 6: $sc[5] = \{3, 4, 8\}$

Linia 7: $sl[6] = \{2, 3, 5, 7\}$

Coloana 7: $sc[6] = \{7, 9\}$

Linia 8: $sl[7] = \{2, 4, 7\}$

Coloana 8: $sc[7] = \{2, 3, 5, 7, 8\}$

			5	8	9	3	6
			3	5	4		
9	3	5	4			7	8
6	7	9		8			5
						8	
	1			7	4		2
7			2	5			3
4					7	2	
					1		

Mulțimile corespunzătoare celor patru blocuri sunt:

(0, 0): $sp[0][0] = \{3, 5, 9\}$

(0, 1): $sp[0][1] = \{3, 4, 5, 8\}$

(0, 2): $sp[0][2] = \{3, 4, 5, 6, 7, 8, 9\}$

(1, 0): $sp[1][0] = \{1, 6, 7, 9\}$

(1, 1): $sp[1][1] = \{4, 7, 8\}$

(1, 2): $sp[1][2] = \{2, 5, 8\}$

(2, 0): $sp[2][0] = \{4, 6, 7\}$

(2, 1): $sp[2][1] = \{1, 2, 5\}$

(2, 2): $sp[2][2] = \{2, 3, 7\}$

Declarăm variabilele globale sl , sc , t și poz cu semnificațile:

- $sl[i], sc[j]$: tablouri unidimensionale ce conțin mulțimile de cifre aflate pe linii, respectiv coloane;
- t : vector ce conține, în ordine, valorile din câmpurile completate inițial;
- poz : vector ce conține, în ordine, pozițiile câmpurilor ne-completate inițial.

Caracterele se citesc sevențial din fișierul de intrare și se procesează corespunzător: dacă ch este *, atunci perechea formată din linia și coloana curente se adaugă la vectorul poz ; dacă este cifră, atunci valoarea sa se adaugă mulțimilor liniei, coloanei și blocului curent. După cîtirea unui astfel de caracter valid, se actualizează linia și coloana curente (care sunt inițializate cu 0): dacă se ajunge la coloana 9 (DIM), atunci ne deplasăm pe linia următoare și inițializăm coloana cu 0: $1++$; $c=0++$.

```

if(isdigit(ch) || v==ch){
    if(V==ch) poz.push_back(TPair(l, c));
    else{
        sl[l].insert(ch-'0');
        sc[c].insert(ch-'0');
        sp[l/NR][c/NR].insert(ch-'0');
        t.push_back(ch-'0');
    }
    if(DIM==++c) {l++; c=0;}
}

```

Vom codifica soluția ca un vector v cu aceeași lungime ca și poz , cu semnificația $v[i]$ este cifra care se scrie la poziția $poz[i]$.

Vom scrie o metodă de tip *Backtracking* recursiv *back()*: pentru nivelul k , se verifică dacă avem vectorul v complet, caz în care se face afișarea. Altfel își se vor atribui lui $v[k]$ toate valorile posibile dintre 1 și 9 și pentru fiecare dintre ele se verifică dacă poate conduce la o soluție cu valorile $v[0], v[1], \dots, v[k-1]$ folosind metoda *isCandidate()* deja fixate. Dacă $v[k]$ este candidat la soluție, atunci valoarea $v[k]$ se adaugă mulțimilor corespunzătoare: linia $v[k].first$, coloana $v[k].second$ și blocul $[poz[k].first/NR, [poz[k].second/NR]$, după care se apelaază metoda recursivă pentru următorul nivel și se elimină $v[k]$ din mulțimile unde s-a adăugat (prin asignarea unei noi valori pentru $v[k]$ nu mai avem nevoie de valoarea precedentă).

Scrierea soluției se va face cu ajutorul metodei *writeSolution()*: câmpurile careului se vor parcurge în ordine și cele din poz se vor asigna cu valorile din vectorul v , iar în final se va apela la metoda *print()* a clasei *TPair* (aceasta printând secvențial cele două vectori).

Program

```

#include <set>

#include <fstream>
#include <algorithm>

using namespace std;

const char V = '*';
const short DIM = 9;
const short NR = 3;

set<int> sl[DIM], sc[DIM];
set<int> sp[DIM/NR][DIM/NR];

vector<short> t;
typedef pair<short, short> TPair;

```

```

vector<TPair> poz;

bool isCandidat(vector<int> v, int k) {
    int ll = poz[k].first;
    int cc = poz[k].second;
    if(sl[ll].find(v[k])!=sl[ll].end()) return false;
    if(sc[cc].find(v[k])!=sc[cc].end()) return false;
    if(sp[ll/NR][cc/NR].find(v[k])!=sp[ll/NR][cc/NR].end())
        return false;
    return true;
}

void writeSolution(vector<int> v, ofstream& out){
    int k=0, j=0;
    int ll=0, cc=0;
    while(k<DIM){
        if(poz[k].first==ll && poz[k].second==cc){
            out<<v[k++]<<" ";
        }
        else out<<t[j++]<<" ";
        if(++cc==DIM) (ll++; cc=0; out<<endl);
    }
    out<<endl;
}

void back(int k, vector<int>& v, ofstream& out){
    if(k==poz.size()) { writeSolution(v, out); return; }
    for(j=0; j< DIM; j++) {
        v[k]=j;
        if(isCandidat(v, k)){
            sl[poz[k].first].insert(v[k]);
            sc[poz[k].second].insert(v[k]);
            sp[poz[k].first/NR][poz[k].second/NR].insert(v[k]);
            if(v[k]==t[j]) {
                sl[poz[k].first].erase(v[k]);
                sc[poz[k].second].erase(v[k]);
                sp[poz[k].first/NR][poz[k].second/NR].erase(v[k]);
            }
        }
    }
}

int main(){
    ifstream in("sudoku.in");
    char t=0, c=0;
    char ch;
    while(in && !in.eof() && in>>ch && t<DIM){
        if(isdigit(ch) || V==ch){
            if(V==ch) poz.push_back(TPair(t, c));
        }
    }
}

```

```

    else{
        s1[l].insert(ch-'0');
        sc[c].insert(ch-'0');
        sp[l/NR][c/NR].insert(ch-'0');
        t.push_back(ch-'0');
    }
    if(DIM==++c) (l++; c=0);
}
}

vector<int> v(poz.size(), 0);
ofstream out("sudoku.out");
back(0, v, out);

return 0;
}

```

Probleme propuse

- Valoarea de pe un câmp inițial completat (l, c) este intersecția mulțimilor liniei $s1[l]$ și a coloanei c $sc[c]$. Folosiți această observație și eliminați din program utilizarea tabloului t (în metoda `writeSolution()` se va scrie în loc de $t[j++]$ valoarea $s1[l]\cap sc[cc]$ – aceasta trebuie să fie o mulțime formată dintr-un singur element).
- Modificați programul astfel încât să nu mai folosească variabile globale.
- Într-o nouă fișier cu extensie `.sud` se conține reprezentarea fișierului de intrare dat. Fișierul de tip `sudoku.in` îl va conține următoarele. Modificați programul, astfel încât să scrie, în această situație mesajul „*nu există soluție*”. Exemplu:

sudoku1.in	sudoku1.out
5***92***71 *6*****8* *51*****7 *4***95*** ***6***** 915*6****4 *1*8*5*3* 23***4*8** *	nu există soluție

- Implementați o metodă de tip *Backtracking* iterativ.
- Scrieți o metodă care verifică dacă un careu complet dat respectă condițiile Sudoku (este corect).
- Paginile Web <http://www.sachsenteck.de/en/index.htm>, http://www.mau.org/edito/mathgames/mathgames_09_05.html conțin mai multe variante ale jocului Sudoku (de exemplu, Shidoku, Rokudoku, Maxi Sudoku, Irregular Sudoku etc.). Scrie-

programe și pentru alte variante; găsiți eventual un algoritm generalizat pentru rezolvarea mai multor tipuri de Sudoku.

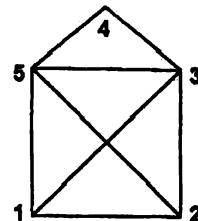
Problema 13. Căsuța lui Moș Crăciun

Copiii (uneori și adulții) desenează de generații căsuța lui Moș Crăciun. Adică acea căsuță obținută prin desenarea patru laturi și a se ridica de pe hârtie creionul și fără a trasa de două ori o linie. Acest lucru este posibil doar atunci când se pornește dintr-unul dintre vârfurile de jos ale căsuței.



Trebuie să scrieți un program care să listeze toate posibilitățile de a desena casa, atunci când se pornește din colțul din stânga-jos (1). Colțurile se vor numera ca în figură. O posibilitate ar fi, de exemplu,

„153125432”, ceea ce înseamnă că se pornește din colțul 1, se desenează o linie până la 5, apoi până la 3 etc.



Soluțiile numerotate și în ordine lexicografică se vor scrie în fișierul *moscraciun.out*.

Exemplu:

.....
Solutia 1: 1 2 3 1 5 3 4 5 2						
.....						
Solutia 26: 1 3 5 2 3 4 5 1 2						
.....						
Solutia 44: 1 5 4 3 5 2 3 1 2						

Analiza problemei și proiectarea soluției

Casa lui Moș Crăciun este, de fapt, reprezentată ca un graf neorientat, în care trebuie să găsim ciclurile euleriene. Pentru aceasta, vom folosi din nou un algoritm de tip backtracking, în care construim succesiiv soluția în tabelul $b[k]$; când a fost stabilită o valoare candidat pentru $b[k]$, vom șterge muchia „desenată” ($b[k-1], b[k]$), vom apela metoda recursiv și apoi vom introduce muchia din nou în graf.

Program

```
#include <iostream>
const a[5][5]={0,1,1,0,1,
               1,0,0,1,1,
               1,0,0,1,1,
               0,1,1,0,1,
               1,1,1,1,0};
```

```

        1,0,1,0,1,
        1,1,0,1,1,
        0,0,1,0,1,
        1,1,1,1,0};

int b[9];
int sol=0;
std::ofstream out("moscraiun.out");

void writeSol(){
    out<<"Solutia " << ++sol << ":" ;
    for(int i=0;i<9;i++)
        out<<b[i]+1<<" ";
    out<<std::endl;
}

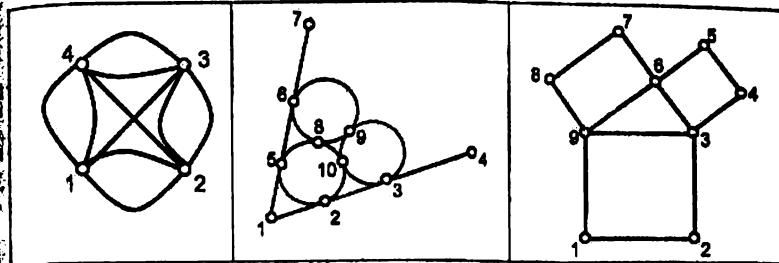
void back(int k){
    int i;
    if(9==k) writeSol();
    else
        for(i=0;i<5;i++)
            if(a[i][b[k-1]]==1 && i!=b[k-1]){
                b[k]=i;
                a[i][b[k-1]]=0;
                a[b[k-1]][i]=0;
                back(k+1);
                a[i][b[k-1]]=1;
                a[b[k-1]][i]=1;
            }
}

int main(){
    b[0]=0;
    back(1);
    return 0;
}

```

Probleme propuse

1. Modificați p. programul astfel încât să rezolve problema în mod iterativ.
2. Scrieți un algoritm de tip *Backtracking* iterativ.
3. Implementați programe și pentru următoarele figuri, pentru a le deseară ca mai sus. În a două figură se începe cu vârful 4, la prima și la a treia cu vârful 1.



Problema 14. Compactarea gabloanelor de test

Se dă o mulțime de teste T ; se cere determinarea unei mulțimi de teste T' de cardinal minim, astfel încât pentru fiecare test din T să existe un test compatibil în T' . Un test este un sir ce conține caractere din mulțimea $S = \{0', '1', 'U', 'Z', 'X\}$. Vom numi S mulțimea de compactare. Exemple de teste: $X01XUZZX01$, $100011ZUX1ZXXX$.

Definiția 1. Caractere compatibile

Spunem că două caractere sunt compatibile atunci când ambele coincid sau când cel puțin unul dintre ele este X (don't care). Vom nota această relație cu „=“ și incompatibilitatea cu „≠“. Atunci când cele două caractere coincid, le vom uni în unul singur (merge), când unul dintre ele este X , atunci rezultatul operației *merge* va fi celălalt caracter (care nu este X). În tabelul următor sunt prezentate relațiile compatibil și

=	0	1	U	Z	X
0	0	*	*	*	0
1	*	1	*	*	1
U	*	*	*	*	*
Z	*	*	*	Z	Z
X	0	1	U	Z	X

Definiția 2. Teste compatibile

Teste date sunt compatibile atunci când lungime și fiecare caracter dintr-un test este compatibil cu caracterul corespunzător din testul celălalt. Din două teste compatibile se creează un test fuzionat (*merge*), în care, pe fiecare poziție, se află caracterul fuzionat din cele două caractere corespunzătoare în cele două teste.

Exemplu: testele $t_1 = 102X0XU$ și $t_2 = X0Z10UU$ sunt compatibile, deoarece $t_1(i) = t_2(i)$, pentru toți $i \in \{1, 2, \dots, 7\}$. Merge(t_1, t_2) = $10210UU$.

Ne dorim apoi ca pentru o mulțime dată de teste să determinăm o mulțime compatibilă de teste, de cardinal minim.

Date de intrare. În fișierul *tests.in*, se găsesc mai multe teste de aceeași lungime, că unul pe linie (cel mult 40, fiecare de lungime maximă 1000).

Date de ieșire. Scrieți în fișierul *tests.out* pe prima linie dimensiunile datelor de intrare (numărul și lungimea testelor), pe a doua linie numărul testelor în mulțimea compatibilă minimală și procentul de reducere a mulțimii inițiale, apoi o linie goală urmată de o mulțime optimă de teste, câte unul pe linie. Exemplu:

<i>tests.in</i>	<i>tests.out</i>
UU0XXZ2U	15 8
XUU0XX11	5 33.3333%
XXUXZXXX	
UUU0ZZ11	UUOX1ZZU
0XX111XX	011111ZZ
XUU0XX11	UUUXZUOU
UOXXXUXU	UUUZZ11U
XXXX1XXX	UUU02Z11
UXXXXZ2U	
011111ZZ	
0XX1XXXX	
XXUXZXXX	
UUUXZUOU	
XUUXZX1X	
OUUXZX11U	

(propusă de Rolf Drechsler, Görschwin Fey, Daniel Große)

Analiza problemei și prelucrarea soluției

Din definiția problemei rezultă următoarele trei propoziții, care ne vor ajuta în găsirea soluției.

Propoziția 1

Dacă $t_0 \neq t_1$ (nu sunt compatibile), $t_0 = t_2$ (compatibile) și $\text{Merge}(t_0, t_2) = t_3$, atunci $t_3 \neq t_1$.

Demonstrație. Deoarece t_0 și t_1 nu sunt compatibile, rezultă că există cel puțin o pozitie k , astfel încât $t_0[k] \neq t_1[k]$. După operația de fuzionare, obținem $\text{Merge}(t_0, t_2) = t_3$ și rezulta că $t_3[k] = t_0[k] \neq t_1[k]$; deci t_3 și t_1 nu sunt compatibile. Dacă ar fi $t_3 = t_1$, rezulta că t_1 și t_3 nu sunt compatibile. □

Propoziția 2

Dacă $t_0 \neq t_1$ și $\text{Merge}(t_0, t_1) = t_2$, $t_2 \neq t_3$ și $\text{Merge}(t_2, t_3) = t_4$, atunci $t_0 \neq t_4$ și $t_1 \neq t_4$.

Demonstrație. Dacă un caracter în t_0 este X , atunci el este compatibil cu caracterul de aceeași poziție în t_4 . Dacă nu este X , atunci prin operația *Merge* a fost creat același caracter și în t_4 . □

Propoziția 3

Dacă $t_0 = t_1$ și $t_1 = t_2$, atunci nu rezultă că $t_0 = t_2$.

Demonstrație. Contraexemplu: $t_0 = 1XU$, $t_1 = XZU$, $t_2 = 0ZX$ (vezi prima poziție!). \square

Pentru rezolvarea problemei, vom utiliza un algoritm de tip Backtracking recursiv. Un vector dat v se va modifica repetat, astfel încât toate perechile de teste compatibile să fie înlocuite cu rezultatul fuzionării lor. Algoritmul se va apela apoi pentru vectorul rezultat. Variabila booleană *compact* verifică dacă există perechi compatibile de teste. Atunci când cel puțin o astfel de pereche există, variabila *compact* va primi valoarea *true* (vectorul încă nu este compact!). Atunci când nu există perechi compatibile în vector, *compact* va rămâne *true*, iar v este una dintre soluțiile posibile. În acest caz, trebuie să mai verificăm dacă vectorul curent v este de dimensiune mai mică decât soluția cea mai bună găsită până acum, salvată în vectorul $vOut$. În acest caz, se va înlocui $vOut$ cu v .

Algoritmul în pseudocod:

```

ALGORITHM_OPTIM_COMPACT_REC( vector v, vector& vOut )
    bool compact
    compact  $\leftarrow$  true
    For ( $i = 1$ ,  $\text{size}(v) - 1$ ; step 1) Execute
        For ( $j = i + 1$ ,  $\text{size}(v)$ ; step 1) Execute
            If ( $t_i$  compatible  $t_j$ ) Then
                compact  $\leftarrow$  false
                 $t \leftarrow \text{merge}(t_i, t_j)$ 
                 $v.replace(t_i, t)$ 
                 $v.add(t_j)$ 
                ALGORITHM_OPTIM_COMPACT_REC(v, vOut)
            End_If
        End_For
        If (compact) Execute
            If ( $\text{size}(vOut) > \text{size}(v)$ ) Then
                 $vOut := v$ 
            End_If
        return;
    End_If
END_ALGORITHM_OPTIM_COMPACT_REC

```

Corectitudinea algoritmului. Algoritmul este corect, pentru că vectorul v este în așa fel manipulat încât proprietățile de compatibilitate se păstrează (înlocuirea a două teste compatibile cu testul lor fuzionat).

Complexitatea algoritmului. Arborele apelurilor recursive are înălțimea maximă (numărul testelor în fișierul de intrare). În acest caz, numărul total de comparații este:

$$\frac{n(n-1)}{2} \cdot \frac{(n-1)(n-2)}{2} \cdot \frac{(n-2)(n-3)}{2} \cdots \frac{2 \cdot 1}{2} = \prod_{k=2}^n \frac{k(k-1)}{2} = \frac{n!(n-1)!}{2^{n-1}}$$

De unde rezultă complexitatea algoritmului: $O(\frac{n!(n-1)!}{2^n})$. □

Vom implementa metodele *compatibleChars()* (verifică dacă două caractere sunt compatibile), *compatibleLines()* (verifică dacă două teste sunt compatibile), *mergeChars()* (fuzionarea a două caractere compatibile) și *mergeLines()* (fuzionează două teste compatibile).

Program

```
#include <fstream>
#include <string>
#include <vector>

using namespace std;

bool compatibleChars(char c1, char c2){
    return (
        (c1 == c2) || // Dacă caracterele sunt compatibile atunci când coincid sau
        (c1 == 'X') || // sunt 'X' și c2 nu este 'X'
        (c2 == 'X') // sau c1 nu este 'X' și c2 coincide cu c1
    );
}

bool compatibleLines(string str1, string str2){
    if(str1.length() != str2.length())
        return false;
    bool retValue = true;
    for(int i=0; retValue && i<str1.length(); i++)
        retValue &= compatibleChars(str1[i], str2[i]);
    return retValue;
}

char mergeChars(char c1, char c2){
    if(!compatibleChars(c1, c2)) return '*';
    char retCh;
    if('X'==c1 && 'X'==c2)
        retCh = 'X';
    } else{
        if(c1=='X') retCh = c1;
        else retCh = c2;
    }
    return retCh;
}
```

```

    else retCh = c2;
}
return retCh;
}

string mergeLines(string str1, string str2){
if(str1.length() != str2.length()) return NULL;
char* str = (char*) malloc(str1.length() + 1);
for(int i=0; i<str1.length(); i++)
    str[i] = mergeChars(str1[i], str2[i]);
str[str1.length()] = '\0';
return str;
}

void recExactCompact(vector<string> v, vector<string>& vOut){
    bool compact = true;
    int i, j;
    string str;
    for(i=0; i<v.size()-1; i++)
        for(j=i+1; j<v.size(); j++)
            if(compatibleLines(v[i], v[j])){
                compact = false;
                str = mergeLines(v[i], v[j]);
                v.erase(v.begin()+i);
                v.erase(v.begin()+j-1);
                v.push_back(str);
                recExactCompact(v, vOut);
            }
    if(compact == true){
        if(vOut.size() > v.size()){
            vOut.clear();
            for(int i=0; i<v.size(); i++) vOut.push_back(v[i]);
        }
    }
    return;
}

double doRecExactCompact(string fIn, string fOut){
ifstream in(fIn.c_str());
ofstream out(fOut.c_str());
string line;
vector<string> v;
int i, j;
while(!in.eof()){
    getline(in, line);
    if(line.size() > 0)v.push_back(line);
}
vector<string> vRet;
}

```

v nu este compact atunci când există două teste (linii) compatibile! Testul fuzionat va înlocui cele două teste găsite compatibile și metoda va fi apelată recursiv. Atunci când nu există teste compatibile, vectorul este compact! Mai verificăm în acest caz și dacă dimensiunea vectorului v este mai mică decât cea a lui vOut, caz în care acesta din urmă se actualizează.

```

for(i=0; i<v.size(); i++) vRet.push_back(v[i]);
recExactCompact(v, vRet);
out << v.size() << " " << v[0].size()
    << endl << vRet.size() << " "
    << vRet.size()*100.0/v.size() << "%" << endl;
out << endl;
for(int i=0; i<vRet.size(); i++){
    out << vRet[i] << endl;
}
return vRet.size()*100.0/v.size();
}

int main(){
doRecExactCompact("tests.in", "tests.out");
return 0;
}

```

Probleme propuse

1. Scrieți următoarele metode:

- Calculează numărul total de perechi compatibile în fișierul de intrare. Exemplu: pentru `test.in` din descrierea problemei, acest număr este 41.
- Calculează proprietățile datelor dintr-un fișier de intrare dat: număr de teste, lungimea lor, distribuția caracterelor pe fiecare coloană.
- Generază fișiere de intrare cu dimensiunile și valoarea `compaction_factor` (gradul compactării) dintre fișierul de intrare și rezultatul așteptat; acesta poate fi folosit ca exemplu pentru model de compacție. Exemplu: fișier de compacție `compaction_factor 8`:

5 linii (teste), 5 coloane (lungimea)	15 linii (teste), 9 coloane (lungimea)
1XUU1	XXXXZXXXX
01111	X1Y1ZYYYYX
Z1UAA	X11111111111
000X0	X1x1Z1111Z
0XZOU	XXXXZX1XX
	XXXXZXXXX
	11x1Z1111Z
	111Y1Z1111Y
	XXX1ZXX1X
	10000U00U0U
	XXXXZXXXX
	X00ZU0ZZU
	XXXXZ11XX
	XXXXZXXXX
	X111ZXXXX

Din cauza complexității sale exponențiale, algoritmul de tip *Backtracking* descris mai sus conduce la un timp de execuție foarte ridicat. Din acest motiv, el poate funcționa doar pentru instanțe ale problemei de dimensiune redusă. În scopul compactării unei mulțimi de teste, se poate utiliza un algoritm de tip *Greedy*, care însă nu va furniza mereu cea mai mică mulțime compatibilă. Pseudocodul pentru un astfel de algoritm este următorul:

```

ALGORITHM_GREEDY_COMPACT
vector v[1..n]
bool compact
compact  $\leftarrow$  false
While (NOT compact) Execute
    If ( $\exists i, t_i : t_i$  compatible  $t_j$ ) Then
        consider first (i, j) lexicographical)
        compact  $\leftarrow$  false
        t  $\leftarrow$  merge(ti, tj)
        v.delete(ti)
        v.delete(tj)
        v.add(t)
    End_If
    Else Execute
        compact  $\leftarrow$  true
    End_Else
END_ALGORITHM_GREEDY_COMPACT

```

Acest algoritm este capabil să prelucreze fișiere de intrare de dimensiune mult mai mare.

- a) Implementați algoritmul de mai sus folosind C++ și Java.
- b) Adăugați programului de mai sus metodele de la problema anterioară și cu algoritmul *Greedy*.
- c) Testați toate aceste metode prin crearea aleatorie de fișiere de intrare și prin compactarea acestora utilizând C++ și Compact. Debutați cu un număr și timpul de execuție pentru metoda *Greedy*. De exemplu, puteți crea automat fișiere de intrare cu următoarele condiții:
 - *compaction_factor* 25;
 - număr de teste de la 100 până la 1000, cu un pas 100 (*f100_xxx.txt* până la *f1000_xxx.txt*);
 - variați și numărul de coloane (lungimea unui test) de la 115 la 915, cu un pas de 200 (*xxx_115.txt* până la *xxx_915.txt*).

Fisierul de ieșire *report.out* poate arăta astfel:

report.out					
file_name	#lines	#cols	#comps	comp_rate	time(sec)
f100_115.txt	100	115	549	36.00%	1
f100_315.txt	100	315	850	24.00%	1
f100_515.txt	100	515	1503	20.00%	1
f100_715.txt	100	715	647	28.00%	1
f100_915.txt	100	915	973	26.00%	2
...					
f600_115.txt	600	115	35915	33.83%	82
f600_315.txt	600	315	29436	30.33%	123
f600_515.txt	600	515	17916	29.83%	163
f600_715.txt	600	715	37335	30.17%	190
f600_915.txt	600	915	8452	26.83%	186
...					
f1000_115.txt	1000	115	16218	32.80%	213
f1000_315.txt	1000	315	89951	33.90%	394
f1000_515.txt	1000	515	28579	29.70%	813
f1000_715.txt	1000	715	258987	17.90%	227
f1000_915.txt	1000	915	86154	16.30%	495

(#comps este numărul de perechi compatibile inițiale)

3. Comparați metodele *Backtracking* recursiv și *Greedy* prin generarea unor fișiere de intrare de diferite dimensiuni și compactarea lor folosind cele două metode. Automatizați procesul, astfel încât să genereze fișiere cu diferite caracteristici, să le compacteze folosind cele două metode, iar rezultatul să fie scris în fișierul *report.out*. Caracteristicile pot să fie, de exemplu, *compaction factor* 50; număr de linii de la 100 la 1000; număr de caractere de la 100 la 1000. Care dintre cele două metode este mai rapidă și rezultatele trebuie scrisă ca mai jos. Marcați cu * cazurile când algoritmul optim de tip *Backtracking* furnizează o soluție mai bună decât *Greedy*. Alternativ factorul de compactare *compaction factor* pentru fișierele de intrare, numărul de linii și numărul de caractere.
- Algoritm Greedy: "Algoritm Backtracking rezultate mai bune decât cel Greedy?"

report.out							
			GREEDY	COMPACT	BACK COMPACT		
			#comps	t (s)	#comps t (s)		
f_5_5.txt	5	5	0	0.00000	0		
f_5_6.txt	5	6	1	80.00%	0		
f_5_7.txt	5	7	4	60.00%	0		
...							
f_11_7_1.txt	7	12	14.44%	0	45.45%	0 **	
f_11_8.txt	11	8	13	45.45%	0	45.45%	0
f_11_9.txt	11	9	4	63.64%	0	63.64%	0
f_12_5.txt	12	5	58	16.67%	0	16.67%	0
...							

f_20_6.txt	20	6	48	55.00%	0	55.00%	0
f_20_7.txt	20	7	31	35.00%	0	35.00%	3
f_20_8.txt	20	8	61	50.00%	0	50.00%	1
"							
f_35_5.txt	35	5	90	54.29%	0	45.71%	395 **
f_35_6.txt	35	6	201	40.00%	0	37.14%	3689 **
f_35_7.txt	35	7	150	40.00%	0	40.00%	2677
f_35_8.txt	35	8	71	57.14%	0	57.14%	82
f_35_9.txt	35	9	116	51.43%	0	51.43%	254

- Testele sunt în program de tip *string*, deși mulțimea de compactare {0, 1, U, X, Z} are doar cinci elemente (aceasta înseamnă că sunt folosiți 8 biți pentru un caracter, când, de fapt, ar fi suficienți trei). Îmbunătățiiți programul astfel încât reprezentarea testelor să fie mai economică, prin folosirea operatorilor pe biți, de exemplu (sau a tipurilor *bitset*, *vector<bool>* din *STL*).
- Dezvoltați programul astfel încât să nu se parcurgă întotdeauna întregul vector pentru găsirea perechilor compatibile de teste. Acestea trebuie doar salvate într-o listă și cu ajutorul ei trebuie prelucrate.

Încă 12 probleme propuse

- Parantetizarea corectă a *n* perechi de paranteze. Scrieți un program care să furnizeze toate parantetizările corecte ale *n* perechi de paranteze. Exemplu:

Tastatură	paranteze.out
<i>n=3</i>	(() () , ((()) , (() () , ((())

6. Câteva informații: Se dă o hartă cu dimensiunea 7×7 și următoarele date de intrare:
- harta.in = fișierul care conține vectorul și mpj.y = fișierul cu numărul vecinilor. Iată urmăriți o modalitate de a colora harta cu un număr minim de culori, astfel încât oricare două ţări vecine să fie colorate diferit. Scrieți pe prima linie a fișierului de ieșire *culori.out* numărul de culori găsit, iar pe următoarea linie lista culorilor corespunzătoare, în ordinea hărții. mpj.y :

harta.in	culori.out
7	4
0 1 1 1 0 0 1	1 2 3 4 1 3 2
1 0 1 1 0 0 0	
1 1 0 1 1 0 1	
1 - 1 0 1 0 1	
0 0 1 1 0 1 1	
0 0 0 0 1 0 1	
1 0 1 1 1 1 0	

Generați toate posibilitățile de a colora harta cu numărul minim de culori.

3. *Produs cartesian.* Produsul cartesian a n multimi A_1, A_2, \dots, A_n este format din toate n -uplele (a_1, a_2, \dots, a_n) cu $a_i \in A_i$. Aceasta se notează cu $A_1 \times A_2 \times \dots \times A_n$:

$$\prod_{i=1}^n A_i = A_1 \times A_2 \times \dots \times A_n = \{(a_1, a_2, \dots, a_n) | a_i \in A_i \forall i = 1, 2, \dots, n\}$$

Numărul total al n -uplelor este $|A_1| \cdot |A_2| \cdot \dots \cdot |A_n|$.

În această problemă, ne dorim determinarea produsului cartesian $\{1, 2, \dots, m\} \times \{1, 2, \dots, n\} \times \dots \times \{1, 2, \dots, n_k\}$. În fișierul *cart.in* se găsesc dimensiunile multimilor. Scrieți toate k -uplele în fișierul *cart.out*, câte unul pe linie. Exemplu:

<i>cart.in</i>	<i>cart.out</i>																
2 3 1 4	<table style="margin-left: auto; margin-right: auto;"> <tr><td>1</td><td>1</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>1</td><td>2</td></tr> <tr><td>—</td><td></td><td></td><td></td></tr> <tr><td>2</td><td>3</td><td>1</td><td>4</td></tr> </table>	1	1	1	1	1	1	1	2	—				2	3	1	4
1	1	1	1														
1	1	1	2														
—																	
2	3	1	4														

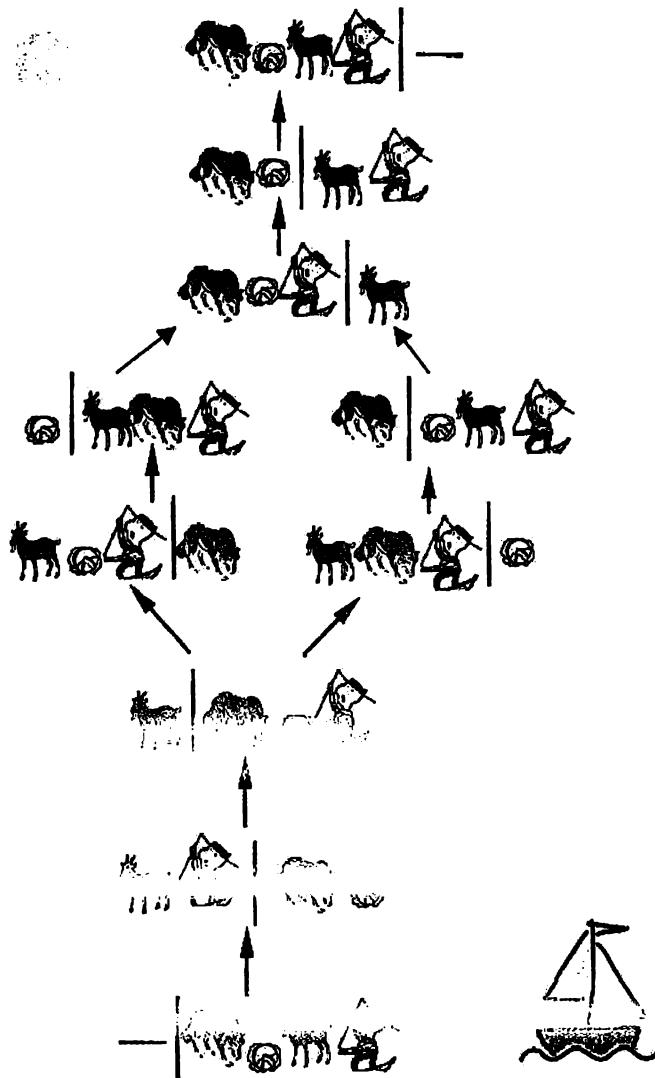
4. *Ce face un țăran cu un lup?* Un țăran se află la drum cu o capră, o varză și un lup și dorește să treacă împreună cu toți însoțitorii un râu. El găsește pe mal o barcă; aceasta este însă atât de mică, încât cu mare greutate începe el și unul dintre însoțitori. Deoarece numai doi încap în barcă și el vrea ca toți să ajungă întregi pe celălalt mal, nu are voie să lase



cumva capra cu lupul sau capra cu varza nesupravegheți, căci capra poate fi

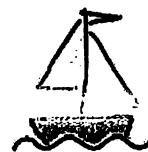
deosebit de săracă și nu poate să le dețină în același loc. Rezolvă problema reprezentată a problemei și rezolvă dilema țăranului folosind metoda *Backtracking*. Câte posibilități există?





Tăranul, lupul, capra și varza

5. **Pioni pe tablă de săk.** Pe o tablă de cămăduță dimensiunea 8×8 țineți pioni. Vezi că unui pion se află în cîmpurile imediat vecine pe orizontală, verticală și diagonala. Pionii sunt aranjați astfel încât formează o singură figură plină (fără orificii), cu



interiorul format numai din pioni), care poate ocupa și pătratele de pe marginea tablei de șah. Să se determine numărul pionilor de pe conturul figurii formate precum și numărul total de pioni cu care aceasta s-a format. Se consideră că un pion este pe contur, dacă numărul vecinilor săi este strict mai mic decât 8.

Date de intrare. Configurația tablei de șah se află în fișierul *pioni.in*. Aceasta are forma a opt șiruri de câte opt caractere, dispuse pe opt linii separate. Pătratele libere sunt codificate prin caracterul „.”, iar cele ocupate prin caracterul „B”.

Date de ieșire. În fișierul de ieșire *pioni.out* se va afișa:

- în cazul unei configurații corecte: pe primul rând, textul *Contur=*, urmat de numărul de pioni care se află pe contur; pe al doilea rând, textul *Total=* urmat de numărul total de pioni din care s-a format figura.
- în cazul când disponerea pionilor nu respectă condițiile problemei (pionii nu constituie figură plină), se va afișa textul *Configuratie incorrecta*.

Exemple:

<i>pioni.in</i>	<i>pioni.out</i>
..... ...BB.. .B.... .BBBBB.. BBBBB.. BBBBB.. .B.B..B... .B.B.. .B...B.P...B.B..	Contur=19 Total=21
....B... .B.B.. .B...B.P...B.B..	Configuratie incorrecta

(Olimpiada Județeană de Informatică, Iași, 2000)

6. *Baze ascunse.* Se dau n numere naturale a_1, a_2, \dots, a_n ($1 \leq n \leq 30$), care au maximum nouă cifre. fiecare număr este reprezentat într-o bază b_i ($2 \leq b_i \leq 10$), astfel încât intervalul $[a, b]$ în care sunt cuprinse toate numerele (transformate în baza 10) are lărgimea minimă*. Datele de intrare sunt date în fișierul *baze.in*. Datele de ieșire sunt date absolută dintre capetele sale.

Date de intrare. Numerele a_1, a_2, \dots, a_n se găsesc în fișierul *baze.in*.

Date de ieșire. Scrieți în fișierul *baze.out* perechile (a_i, b_i) și pentru fiecare reprezentarea în baza 10, ca în exemplu. Pe ultima linie scrieți și intervalul în care se găsesc aceste numere. Exemplu:

base.in	base.out
12102 34215 2314 28756 1231	n Baza(n) n in zecimal
1010101 23413 28457 343421	12102 6 1766
	34215 6 4835
	2314 9 1714
	28756 9 19572
	1231 10 1231
	1010101 4 4369
	23413 5 1733
	28457 9 19330
	343421 5 12361
Interval: [1231, 19572]	

(Olimpiada Națională de Informatică, Timișoara, 1997, exerț modificat)

Indicație: Putem implementa o metodă de tip Backtracking recursiv de genul:

```
void back(int k){
    int i;
    if(k==n)
        processThisCombination();
    else
        for(i=base[k]; i<=10; i++)
            x[k]=i;
            back(k+1);
}
```

Dacă:
 - a fost găsită o configurație de bază ($k=n$), atunci verificăm dacă intervalul corespunzător este mai scurt decât cel mai bun găsit până acum;
 - încă nu a fost găsită o configurație de bază ($k < n$), atunci se setează $x[k]$ cu toate bazele posibile și se apelează metoda pentru nivelul următor ($k+1$) (base[k] este valoarea minimă posibilă a bazei pentru $x[k]$).

Implementați programul fără a folosi variabile globale.

7. *Labyrințul.* Un labirint este reprezentat cu ajutorul unei matrice L , de dimensiune $m \times n$, care conține elemente 0 și 1. Pereții sunt notați cu 0, iar locurile libere cu 1. O persoană se găsește în labirint pe un loc liber. Găsiți toate drumurile care duc afară din labirint, mergând numai pe locuri libere. La un moment dat, se poate

în exemplu:

labirint.in	labirint.out
4 7	...
0 0 1 1 0 0 0	Solutia 3:
0 0 1 1 0 0 0	0 0 1 ..
0 0 1 1 1 1 0	0 0 0 3 0 1 0
0 1 1 1 0 0 0	0 0 1 2 1 1 0
3 3	0 1 1 1 0 0 0
	...

8. *Steagurile.* Ne imaginăm că avem la dispoziție stofe de trei culori: alb, galben, portocaliu, roșu, albastru, verde și negru. Se cere determinarea tuturor

posibilităților de a crea steaguri cu trei culori pe orizontală, astfel încât culorile să fie diferite și în mijloc să fie alb, galben sau portocaliu. Scrieți toate posibilitățile în fișierul *steaguri.out*, câte una pe linie:

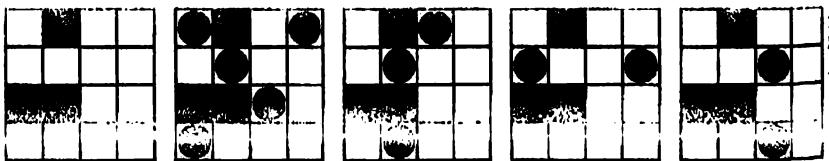
<i>steaguri.out</i>
galben alb portocaliu
...
albastru galben rosu
...
negru portocaliu verde

Fără program, doar pe hârtie, câte steaguri se pot crea?

9. Câte resturi poți să îmi dai? Găsiți toate posibilitățile de a plăti suma S dată cu monede diferite de valori m_1, m_2, \dots, m_n . În fișierul *plata.in* se găsește suma S pe prima linie, urmată de valorile monedelor ($0 \leq S \leq 1200, 1 \leq m_i \leq 50$). Toate modalitățile se vor scrie în fișierul *plata.out*. Exemplu:

<i>plata.in</i>	<i>plata.out</i>
136 12 1 5 25	Solutie: $4 \times 2 + 11 \times 12$... Solutie: $1 \times 1 + 2 \times 5 + 5 \times 25$

10. *Ecuție cu trei necunoscute.* Scrieți în fișierul *ecuatie.out* toate tripletele $(x, y, z \in \mathbb{N} | 2xy + xyz + 4z = 152)$. Generalizați problema.
11. *Ture pe o altfel de tablă de șah.* În șah, tura este piesa care poate să mutată oricăre pătrate vertical și oricăre pătrate orizontal. În această problemă vom considera o tablă de șah de dimensiune mică (cel mult 4×4) ce poate conține și ziduri peste care nu se poate trece. Scopul este plasarea, cât mai multor ture pe tablă astfel încât să nu se poată capătura tută pe alta. O configurație de ture se consideră locul dacă două ture nu sunt pe același rând orizontal sau vertical sau, dacă sunt, ele trebuie să fie separate printr-un zid. Următoarea imagine prezintă cinci stări ale unei tablă de șah.



Prima este o tablă fără ture, a doua și a treia reprezintă configurații corecte, a patra și a cincea reprezintă configurații incorecte. Pentru această tablă, numărul maxim de ture ce pot fi amplasate este 5, iar figura 2 este o modalitate de amplasare.

Date de intrare. În fișierul *ture.in* se găsesc mai multe configurații, în forma următoare: pe o linie, un număr natural n ($1 \leq n \leq 4$) reprezentând dimensiunea tablei; pe următoarele n linii, tabla de șah exprimată prin caractere.

Date de ieșire. În fișierul *ture.out* trebuie să se scrie, pentru fiecare set de date din fișierul de intrare, numărul maxim de ture ce pot fi amplasate pe o tablă corespunzătoare. Exemplu:

<i>ture.in</i>	<i>ture.out</i>
1	1
.	0
2	1
XX	3
XX	6
2	4
X.	4
X.	
3	
...	
...	
...	
4	
....	
..X.	
...	
...	
4	
XXX.	
....	
.X..	
3	
.X.	
..X	
.X.	

12. *Tanța florărescă.* Doamna Tanță este o florărescă grăbită, iar în unele zile se încurcă cu florile sale și nu știe cum să rezolve așezarea lor în cutii. Dimineață, toate florile sunt amestecate în cele, să zicem, n cutii. Tânță are n categorii de flori și ar dori să le sorteze, fiecare tip de floare în cutia sa, printr-un număr minim de mutări. Ea își iubește florile și le mută cu mare grija, câte una; deci, prin *mutare* înțelegem mutarea unei singure flori în altă cutie. Va trebui să-i daiți o mână de ajutor, căci

doamna Tanță este o persoană bună și vă va răsplăti cu un buchet minunat. Doamnelor Tanță nu-i place să aibă flori cu aceleși inițiale, așa că florile sale încep cu litere diferite. Ea are cele mai frumoase flori, dar niciodată mai mult de 10 varietăți. Presupunem că stările inițiale ale cutiilor se află în fișierul *flori.in*. Pentru fiecare instanță, avem pe prima linie numărul *n* de cutii și de soluri de flori, pe a doua linie, sirul reprezentând inițialele florilor așa cum sunt ele considerate la început în fiecare cutie. Apoi, urmează un sir cu $N \times N$ elemente.

Primele *N* elemente reprezintă așezarea florilor în prima cutie, următoarele *N* elemente reprezintă așezarea florilor în a doua cutie etc. Datele de intrare termină cu numărul 0. Va trebui să scrieți un program care memorează în fișierul *flori.out* un raport referitor la aceste instanțe; pentru fiecare dintre ele se vor scrie numărul cazului analizat, numărul minim de mutări, ce fel de floare și câte din fiecare se află în final în fiecare cutie, ca în exemplul următor.

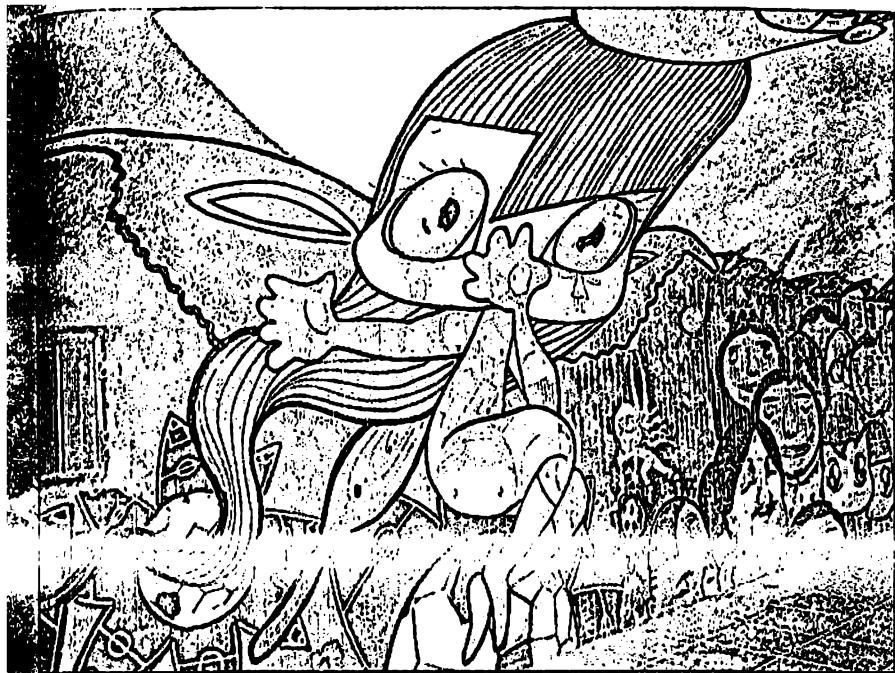
Observație:

Dacă sunt mai multe modalități de mutări cu obținerea unui număr minim, se va considera numai acela cu sirul inițialelor florilor cel mai mic în ordinea alfabetice.

Exemplu:

<i>flori.in</i>	<i>flori.out</i>
<pre> 3 1 10 5 20 10 5 20 20 10 4 TCDF 4 6 7 8 9 1 2 4 3 9 90 7 8 4 5 1 GOLITA 21 14 1 8 9 70 43 1 2 3 5 4 7 43 34 12 65 7 89 76 12 13 31 4 12 6 </pre>	<pre> Cazul 1: Flori: A 11; B 5 Numar minim de mutari: 50 C 20 B 35 G 40 Cazul 2: Flori: A 11; 4 Numar minim de mutari: 57 F 20 T 24 D 104 C 20 Numar cutii: 5 Numar minim de mutari: 348 O 139 G 120 A 134 T 114 L 47 </pre>





Graffiti in München

Capitolul 11

Programarea dinamică

14 probleme complet rezolvate, 54 de probleme propuse

- Fundamente și proprietăți ale metodelor
- Numărarea lepurașilor
- Subșir crescător maximal
- Cel mai lung subșir comun (LCS)
- Triunghiul de numere
- Domino
- Împărțirea cadourilor
- Sume asemănătoare
- Scoțienii la Oktoberfest
- Calul pe tabla de șah
- Flutuările la biserici
- Sume de produse
- Triangularizare minimală a unui poligon convex
- Înmulțirea unui țir de matrice
- *Edit-Distance*

Magia se ascunde întotdeauna în detaliu.

Theodor Fontane

Fundamente și proprietăți ale metodelor

1. Apariția conceptului. Programarea dinamică este o metodă de programare care rezolvă multe probleme de optimizare. Termenul a fost introdus în 1940 de către matematicianul american Richard Bellman (1920-1984). La început a fost folosit în teoria controlului și în acest domeniu se vorbește despre principiul lui Bellman al programei dinamice.

2. Principiul optimalității. Metoda programării dinamice se bazează pe principiul optimalității. Aceasta în-



Richard Bellman

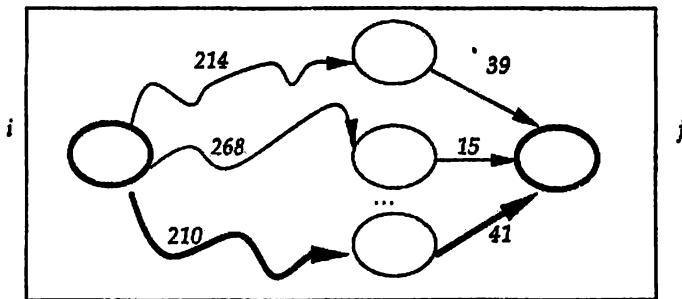
©IEE History Center

1970 I.M. Zemans Reception Lecture

[http://www.ieee.org/organizations/
history_center/legacies/bellman.html](http://www.ieee.org/organizations/history_center/legacies/bellman.html)

sigură că rezolvă problema dată în mod optim, ceea ce înseamnă că căutăm soluții mai mici, iar unele dintre soluțiile acestor subprobleme participă la determinarea soluției optime pentru problema dată. Principiul optimalității este caracteristic și subproblemelor.

Să ne imaginăm că avem să determinăm un drumeu minim de la un nod i la un alt nod j ale unui graf. Pentru aceasta, calculăm drumurile minime de la i la toate vârfurile vecine lui j . Pe baza acestora și pe baza costurilor muchiilor de la vecini la vârful j , determinăm drumul minim de la i la j . Determinarea lungimii drumului de la i la vecinii lui j sunt subprobleme ale problemei date.



Determinarea drumului minim dintre două vîrfuri ale unui graf

De obicei, soluția unei probleme asupra căreia se aplică principiul optimalității este bazată pe trei pași:

1. împărțirea problemei în probleme de dimensiuni mai mici;
2. rezolvarea optimă a subproblemelor tot pe baza acestui model în trei pași;
3. combinarea soluțiilor subproblemelor pentru găsirea soluției optimale a problemei date.

3. Suprapunerea problemelor și salvarea soluțiilor optimale ale subproblemelor (memoization). Alte două caracteristici ale metodelor sunt suprapunerea și salvarea soluțiilor optimale ale subproblemelor. Suprapunerea este o tehnică de calcul care rezolvă o problemă și o conduce la determinarea soluției optimale din multe probleme de dimensiuni mai mari. Memoization înseamnă că soluțiile subproblemelor sunt salvate pentru a fi folosite ulterior, ceea ce îmbunătățește semnificativ timpul de execuție.

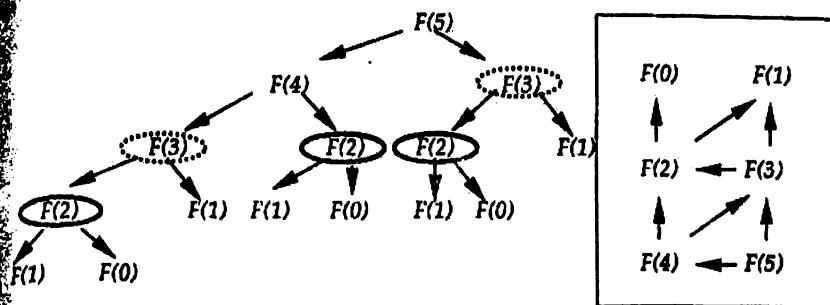
4. Exemplu introductiv: sirul lui Fibonacci. Considerăm problema calculării termenilor din sirul lui Fibonacci, care este definit astfel:

- $F(0) = 0, F(1) = 1$
- $F(n) = F(n - 1) + F(n - 2)$ pentru toți $n > 2$

(1)

Pentru a calcula $F(5)$, avem nevoie de numerele $F(4)$ și $F(3)$.

În general, pentru a putea calcula $F(n)$ avem nevoie de $F(n - 1)$ și $F(n - 2)$. Dacă aceste numere ar fi calculate independent – de exemplu, cu un algoritm recursiv –, ar fi să rezultă că lucru să rezolvăm de nou în fiecare oar considerând $F(5)$:



Calculul lui $F(5)$: arborele recursiv și digraful aciclic (engl., Directed acyclic graph, DAG)

În cazul recursiv, am calculat $F(2)$ de trei ori și $F(3)$ de două ori, fiecare independent. O altă metodă ar fi calcularea valorilor $F(2), F(3), F(4), \dots, F(n)$ și salvarea rezultatelor. Formal, cele două variante se pot scrie:

Algoritmii recursiv și iterativ pentru calcularea numerelor din sirul lui Fibonacci

Determinarea recursivă a celui de-al n -lea termen Fibonacci	Determinarea iterativă a celui de-al n -lea termen Fibonacci (programare dinamică)
$\text{Fib}(n)$ If ($n = 0$ or $n = 1$) Then <i>return</i> 1 Else Execute <i>return</i> $\text{Fib}(n - 1) + \text{Fib}(n - 2)$ End_Else	$\text{Fib}(n)$ $\text{F}[0] \leftarrow 0$ $\text{F}[1] \leftarrow 1$ For ($i = 2$; $i \leq n$; $i = i + 1$) Execute $\text{F}[i] \leftarrow \text{F}[i - 1] + \text{F}[i - 2]$ End_For <i>return</i> $\text{F}[n]$

Folosim un program pentru a compara ambii algoritmi, care afișează și timpii de execuție pentru fiecare metodă în parte.

Comparație a timpilor de execuție pentru algoritmii Fibonacci recursiv și iterativ

N	F(n)	Timp de execuție -recursiv-	Timp de execuție -iterativ-
34	9227465	3	<1
35	14930352	5	<1
36	24157817	7	<1
37	39088169	11	<1
38	63245986	18	<1
39	102334155	30	<1
40	165580141	46	<1
41	267914296	76	<1
42	433494437	122	<1
43	701408733	198	<1
44	1134903170	320	<1
45	1836311903	522	<1
46	2971215073	845	<1
47	4807526976	1361	<1
48	7778742049	2211	<1
49	12586269025	3710	<1
50	20365011074	5904	<1

În această tabelă, se remarcă faptul că timpul de execuție pentru varianta recursivă crește extrem de repede, din cauză că fiecare număr este calculat independent. În acest caz, complexitatea algoritmului este exponențială: $O(2^n)$. A doua variantă folosește principiul optimării. Fiecare număr este calculat numai o dată și valoarea lui este salvoată într-un vector rezultat, ceea ce reduce complexitatea la $O(n)$, unde n este numărul de calculuri.

5. Bottom-up vs top-down. De obicei, metoda folosește una dintre următoarele strategii:

dinainte. După aceea, soluțiile lor vor fi folosite pentru a determina soluțiile optimale ale unor subprobleme de dimensiune mai mare. De obicei, se completează o tabel cu soluțiile subproblemelor și, la sfârșit, soluția problemei inițiale se găsește în aceasta. Avantajul este că menevrarea soluțiilor în tabel este foarte rapidă.

Top-down. Problema se împarte în subprobleme de dimensiune mai mică, ce sunt rezolvate, iar soluțiile sunt salvate pentru cazul când ele vor fi necesare. O subproblemă se rezolvă o singură dată.

6. Comparări cu alte metode de programare. Altfel decât în cazul metodei *Divide et Impera*, unde subproblemele trebuie să fie independente una de alta, în cazul programării dinamice, poate exista situația când o subproblemă participă la

rezolvarea mai multor subprobleme de dimensiuni mai mari. De aceea, o subproblemă trebuie rezolvată doar o singură dată, iar soluția trebuie salvată, pentru a putea fi utilizată în rezolvarea mai multor probleme de dimensiuni mai mari (*memoizare*).

Metoda programării dinamice nu este o tehnică standardizată, cum este, de exemplu, *Backtracking*. Proiectarea algoritmului trebuie să satisfacă doar câteva principii generale. De aceea utilizarea metodei este uneori foarte dificilă și ea presupune deseori cunoștințe solide de matematică.

Deoarece soluția se determină fără a lua în considerare toate cazurile posibile, putem găsi asemănări și cu metoda *Greedy*. Diferența este însă aceea că, în cazul *Greedy*, se adaugă la soluția parțială un optim local, iar soluția nu este, de obicei, cea optimă, pe când programarea dinamică folosește principiul optimalității și conduce la o soluție optimă.

Probleme propuse

- Definiți conceptele: principiul optimalității, suprapunerea problemelor, *memoizare*.
- Care este procedura de lucru a metodei?
- Un alt exemplu tipic pentru folosirea metodei este calcularea coeficienților binomiali:

$$\begin{aligned} C_n^k &= 1, \text{ dacă } k = 0 \text{ sau } n = k \\ C_n^k &= C_{n-1}^{k-1} + C_{n-1}^k, \quad n > k, \quad k \neq 0 \end{aligned}$$
 (2)

Demonstrați următoarele recunoscători și de gradului acelui pentru determinarea valorii C_4^2 (vezi figura pentru calculul lui $F(5)$).

Problema 1. Numărarea iepurașilor

În jurul anului 1180 în Pisa. Deoarece tatăl său făcea comerț cu țările nord-africane, Fibonacci a învățat cifrele hindu-arabe și metodele de calcul ale matematicienilor arabi. El a scris cartea *Liber Algorismi* (Cartea Arăbicului), în care reconinândă folosirea cifrelor hindu-arabe și prezintă probleme matematice spectaculoase, care mai târziu vor fi mereu reluate de către alți autori. De exemplu:

Cineva are 22 de perechi de iepuri între grădini și încuijuri în de un zid. Câte perechi de iepuri se vor naște în fiecare an,



dacă se presupune că, în fiecare lună, fiecare pereche dă naștere unei alte perechi și că perechile de iepuri sunt fertile la două luni după naștere?

Șirul care se deduce din această problemă, cunoscut ca *șirul lui Fibonacci*, a fost deja prezentat în prima parte a capitolului: $F_0 = 0$, $F_1 = 1$ și, mai departe, $F_{n+1} = F_n + F_{n-1}$. Așadar, primii termeni ai șirului sunt: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377.

Scrieți un program care să calculeze cu ajutorul algoritmilor recursiv și iterativ primii 50 de termeni ai șirului și să afișeze în plus și timpii de execuție în secunde pentru fiecare dintre cele două metode:

N	FibRec(N)	Timp(FibRec(N))	FibIt(N)	Timp(FibIt(N))
33	3524578	t1= 2	3524578	t2= 0
34	5702887	t1= 3	5702887	t2= 0
35	9227465	t1= 4	9227465	t2= 0
36	14930352	t1= 7	14930352	t2= 0
37	24157817	t1= 11	24157817	t2= 0
.....

Analiza problemei și proiectarea soluției

Vom scrie metodele *fib01()* și *fib02()*, care implementează cei doi algoritmi. Prima metodă determină fiecare termen independent de alți termeni. A doua metodă construiește iterativ un vector *v* și, pentru calcularea fiecărei noi valori *v[i]*, folosește termenii *v[i-1]* și *v[i-2]* deja calculați. Pentru determinarea timpului, folosim funcția *time()* din anexul *ctime*. Această funcție salvează timpul într-o variabilă de tip *time_t*, unde este de tipul structurii *timeval*. Funcția *time()* returnează timpul de la anumită perioadă 01.01.1970, 00:00, până la 18.01.2038, 00:00. Funcția *time()* are sintaxa: *time_t time(time_t *timp_in_secunde)* și returnează timpul-sistem. Dacă funcția are ca parametru adresa unei variabile de tip *time_t*, atunci timpul se va salva în această variabilă. Dacă se dă ca parametru NULL, atunci nu mai are loc acesta salvare.

Program

```
#include <vector>
#include <ctime>

using namespace std;

const int n = 50;

unsigned long long fib01(int n){
    if(n<=1) return n;
    return fib01(n-1) + fib01(n-2);
}

unsigned long long fib02(int n){
    std::vector<long long> v;
    v.push_back(0);
    v.push_back(1);
    for(int i=2; i< n; i++)
        v.push_back(v[i-1]+v[i-2]);
    return v[n];
}
```

```

int main(){
    unsigned i;
    time_t start, stop;
    ofstream out("FiboReport.txt",
                 ios::app);
    for(i=0; i<=n; i++){
        out << endl;
        out.width(4); out << i;
        start = time(NULL);
        out.width(15);
        out << fibo1(i);
        stop = time(NULL);
        out << " t1=";
        out.width(6);
        out << stop-start;
        start = time(NULL);
        out.width(15);
        out << fibo2(i);
        stop = time(NULL);
        out << " t2=";
        out.width(6);
        out << stop-start;
    }
    return 0;
}

```

Probleme propuse

- Şirul definit prin $L_0 = 2$, $L_1 = 1$ şi $L_n = L_{n-1} + L_{n-2}$ se numeşte şirul lui Lucas. Deseori numerele lui Lucas se pot exprima elegant cu ajutorul unor sume ale numerelor Fibonacci. Modificaţi programul de mai sus astfel încât să calculaţi numerele lui Lucas în locul celor ai lui Fibonacci.
- Aşa cum am specificat, între şirul lui Lucas şi cel al lui Fibonacci există multe relaţii, cum ar fi: $L_{2n+2} \times (-1)^{n-1} = 5F_n^2$. Scrieţi un program care verifică această inegalitate pentru $n = 0, 1, 2, \dots, 25$ şi listează rezultatele într-un fişier, de exemplu, în forma:

	$2(-1)^{(2n+2)}$	$(F_n)^2$	
0	0	0	OK!
1	5	5	OK!
2	5	5	OK!
22	1568397605	1568397605	OK!
23	4106118245	4106118245	OK!
24	10749957120	10749957120	OK!
25	28143753125	28143753125	OK!

- Şirul definit prin $G_0 = 0$, $G_1 = 1$, $G_2 = 2$ (şi $G_3 = 4$) şi $G_n = G_{n-1} + G_{n-2} + G_{n-3} + G_{n-4}$ se numeşte şirul lui Tribonacci (respectiv Quadranacci). În general, au loc pentru termenii de bază $G_i = 2^{i-1}$ pentru $i > 0$ şi $G_0 = 0$. Scrieţi un algoritm care calculează al n -lea termen pentru şirul K -bonacci:

$$G_0 = 0, G_1 = 1, \dots, G_{k-1} = 2^{k-1}$$

$$G_n = G_{n-1} + G_{n-2} + \dots + G_{n-k} \text{ pentru } n \geq k$$

Scrieți un program care pentru numerele naturale k și n date afișează valoarea $G_{k,n}$, care se încadrează în tipul *unsigned long long*. Folosiți metoda programării dinamice. Exemplu:

kBonacci.in	kBonacci.out																						
3 50	<table> <thead> <tr> <th>n</th> <th>$k\text{Bonacci}(3, n)$</th> </tr> </thead> <tbody> <tr><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td></tr> <tr><td>2</td><td>2</td></tr> <tr><td>3</td><td>3</td></tr> <tr><td>4</td><td>6</td></tr> <tr><td>...</td><td></td></tr> <tr><td>47</td><td>1424681173049</td></tr> <tr><td>48</td><td>2620397211992</td></tr> <tr><td>49</td><td>4819661885417</td></tr> <tr><td>50</td><td>8864740270458</td></tr> </tbody> </table>	n	$k\text{Bonacci}(3, n)$	0	0	1	1	2	2	3	3	4	6	...		47	1424681173049	48	2620397211992	49	4819661885417	50	8864740270458
n	$k\text{Bonacci}(3, n)$																						
0	0																						
1	1																						
2	2																						
3	3																						
4	6																						
...																							
47	1424681173049																						
48	2620397211992																						
49	4819661885417																						
50	8864740270458																						
5 57	<table> <thead> <tr> <th>n</th> <th>$k\text{Bonacci}(5, n)$</th> </tr> </thead> <tbody> <tr><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td></tr> <tr><td>2</td><td>2</td></tr> <tr><td>3</td><td>4</td></tr> <tr><td>4</td><td>8</td></tr> <tr><td>5</td><td>15</td></tr> <tr><td>...</td><td></td></tr> <tr><td>55</td><td>7280158119374827</td></tr> <tr><td>56</td><td>14312614013268229</td></tr> <tr><td>57</td><td>28625231385564558</td></tr> </tbody> </table>	n	$k\text{Bonacci}(5, n)$	0	0	1	1	2	2	3	4	4	8	5	15	...		55	7280158119374827	56	14312614013268229	57	28625231385564558
n	$k\text{Bonacci}(5, n)$																						
0	0																						
1	1																						
2	2																						
3	4																						
4	8																						
5	15																						
...																							
55	7280158119374827																						
56	14312614013268229																						
57	28625231385564558																						

Scrieți și o metodă recursivă și comparați timpii de execuție.

4. Un alt exemplu clasic de programare dinamică este calcularea coeficienților binomiali C_n^k , care sunt definiti și mai sus în formula (2). Să de această dată, algoritmul iterativ este mult mai eficient decât algoritmul recursiv. Implementați ambele variante și comparați timpii de execuție pentru diferite valori. Afipați într-o formă corespunzătoare și triunghiul lui Pascal până la un nivel n dat. Exemplu pentru $n = 3$:

0		1		
1		1	1	
2	1	2	1	
3	1	3	3	1

Problema 2. Subșir crescător maximal

Propoziție (Erdős, Szekeres). Într-un șir $a_1, a_2, \dots, a_{mn+1}$ cu $mn + 1$ numere reale diferite, există întotdeauna un subșir crescător

$$a_{i_1} < a_{i_2} < \dots < a_{i_{m+1}} \quad (i_1 < i_2 < \dots < i_{m+1})$$

de lungime $m + 1$, un subșir descrescător

$$a_{j_1} > a_{j_2} > \dots > a_{j_{n+1}} \quad (j_1 < j_2 < \dots < j_{n+1})$$

de lungime $n + 1$ sau amândouă. O demonstrație a acestei propoziții se poate construi, de exemplu, folosind metoda reducerii la absurd și principiul lui Dirichlet. □

Se cere determinarea subșirului de lungime maximă într-un șir dat cu numere întregi.

Date de intrare. În fișierul *ascending.in* se găsește un șir cu numere întregi, maxim 2000 și fiecare mai mic în modul decât 20.000.

Date de ieșire. Scrieți încă o dată șirul dat în fișierul *ascending.out*, urmat de subșirul crescător maximal, ca în exemplu:

ascending.in	ascending.out
<pre>3 5 76 1 45 2 31 89 90 0 4 15 23 47 95 21 67 8 13 11 5 145 132 77</pre>	<pre>--- Input data: Length: 24 Elements: 3 5 76 1 45 2 31 89 90 0 4 15 23 47 95 21 67 8 13 11 5 145 132 77 --- Output data: Maximal ascending substring: 1 2 4 15 23 47 95 145 Length: 8</pre>

Analiza problemei și proiectarea soluției

Construim vectorul a ce conține șirul dat în fișierul de intrare și apoi vectorii $vPred$ și v cu semnificațiile:

- $v[i] \leftarrow 1 + \max\{v[j] \mid j < i \text{ și } a[j] < a[i]\}, i = 1, \dots, n - 1$
 - $vPred[0] \leftarrow -1$ (primul element nu are nici un predecesor)
- Dacă $\exists j_{max}$ a.î. $v[j_{max}] = \max\{v[j] \mid j < i \text{ și } a[j] < a[i]\}$, atunci $vPred[i] = j_{max}$,

$v[i]$ reprezintă lungimea subșirului maximal care îl are pe $a[i]$ ca ultim element.

Subproblemă. Fie sirul a_1, a_2, \dots, a_n dat. Determinați cel mai lung subșir al său care îl are pe a_n ca ultim element.

$v_{Pred}[i]$ este indexul predecesorului elementului a_i în acest subșir crescător maximal cu a_n ca ultim element. Acest vector ne va ajuta să reconstruim recursiv subșirul cu ajutorul funcției *recoverSubstring()*.

Program

```
#include <fstream>
#include <vector>

using namespace std;

vector<int> vPred;
vector<int> a, v;
int imax;

void readData(ifstream& in){
    int aux;
    while(in && !in.eof()){
        in >> aux;
        a.push_back(aux);
    }
}

void recoverSubstring(int i, ofstream& out){
    if(vPred[i]+1) recoverSubstring(vPred[i], out);
    out << a[i] << " ";
}

void process(){
    int j, i, n;
    v.push_back(-1);
    v.rend.push_back(-1);
    imax = 0;
    n = (int)a.size();
    for(i=1; i<n; i++){
        v.push_back(1);
        if(a[i]<a[i-1]) v[i] = -1;
        for(j=0; j<i; j++)
            if(a[j]<a[i]&&v[j]+1>v[i])
                {
                    v.pop_back();
                    v.push_back(v[j]+1);
                    vPred.pop_back();
                    vPred.push_back(j);
                }
    }
}
```

Această buclă *for* poate fi scrisă și astfel (folosim în blocul *if* operatorul de acces la elementele vectorului *v*):

STL *push_back* și *pop_back*:

```
if(a[j]<a[i]&&v[j]+1>v[i]){
    v[i] = v[j] + 1;
    vPred[i] = j;
}
```

```

        if(v[i]>v[imax]) imax=i;
    }

}

void writeData(ofstream& out){
    out << "---- Input data: " << endl;
    out << "Length: " << (int)a.size() << endl;
    out << "Elements: ";
    for(int i=0; i<(int)a.size(); i++)
        out << a[i] << " ";
    out << endl;
    out << "---- Output data: " << endl;
    out << "Maximal ascending substring: ";
    recoverSubstring(imax, out);
    out << endl << "Length: " << v[imax] << endl;
}

int main(){
    ifstream in("ascending.in");
    ofstream out("ascending.out");
    readData(in);
    process();
    writeData(out);
    return 0;
}

```

Probleme propuse

Dezvoltați programul să-l încălță să fie scris și singurul descreșcător maxim în fișierul de ieșire:

substring.in	substring.out
31 89 90 0 4 15 23 47 95 21 67 8 13 11 5 145 132 77 -1 -2 57	Length: 24 Elements: 3 5 76 1 45 2 31 89 90 0 4 15 23 47 95 21 67 8 13 11 5 145 132 77 --- Output data: Maximal descending substring: 3 2 1 0 45 23 47
	Length: 8 --- Maximal descending substring: 76 45 31 23 21 13 11 5 -1 -2 Length: 10

2. Se poate să existe mai multe subșiruri cu aceeași lungime maximală. Modificați programul astfel încât să fie afișate toate aceste subșiruri.
3. Scrieți un nou program ce „verifică” propoziția lui Erdős și Szekeres pentru mai multe seturi de date. Acesta trebuie să funcționeze astfel: se generează aleatoriu P numere naturale, pentru orice m, n cu proprietatea că $m \leq n$ și $P = mn + 1$, se determină subșirurile maximale crescător și descrescător. În final, se verifică propoziția.
4. *Cuvinte potrivite.* Într-un fișier se găsește un text scris pe mai multe linii, format din cuvinte ce folosesc doar literele mici ale alfabetului englez. Scrieți un program care determină subșirul maximal de cuvinte din text pentru care este îndeplinită condiția că ultimele două litere ale unui cuvânt din sir coincid cu primele două litere din cuvântul următor.

Date de intrare. În fișierul *cuvinte.in* se găsesc cel mult 1000 de cuvinte (fiecare de lungime minimă 2 și maxim 100), care sunt separate prin spații sau sfârșit de linie (*newline*).

Date de ieșire. Scrieți pe prima linie a fișierului *cuvinte.out* numărul de cuvinte din subșirul găsit și, începând cu linia următoare, cuvintele în ordine, câte unul pe linie. Exemplu:

<i>cuvinte.in</i>	<i>cuvinte.out</i>
griul	7
visului ultim	griul
cazu imprecis in	ultim
clipa aceea de isterie	imprecis
în cîteva zile	in
te vechi și tenebroase	isterie
	în
	vechi
	tenebroase

Problema 3. Cel mai lung subșir comun (LCS)

Problema determinării celor mai lung subșir comun (longest common subsequence, LCS) este o problemă clasică de programare dinamică.

Fie $X = (x_1, x_2, \dots, x_m)$ și $Y = (y_1, y_2, \dots, y_n)$ două siruri de numere întregi cu n , respectiv m elemente. Se cere determinarea unui subșir maximal comun al celor două date.

mai lung subșir comun este $Z = [4, 10, 2, 0]$ și are lungimea 4.

Date de intrare. În fișierul *lcs.in* se află cele două siruri date, pe prima linie m și n și pe următoarele linii valorile elementelor sirurilor ($1 \leq m, n \leq 500$) de tip *int*.

Date de ieșire. Scrieți un subșir comun maximal în fișierul *lcs.out*, în forma din exemplu

lcs.in	lcs.out
<pre> 8 9 10 4 20 10 40 2 0 60 4 90 7 10 70 2 71 81 0 14 22 34 5 67 8 8 9 0 12 3 45 6 78 91 231 12 34 45 67 79 57 8 321 55 0 33 12 1 2 3 44 45 56 91 21 22 23 </pre>	<pre> Lungimea maxima: 4 4 10 2 0 Lungimea maxima: 8 34 67 8 0 12 3 45 91 </pre>

Analiza problemei și proiectarea soluției

Vom nota $X_m = \{x_1, x_2, \dots, x_m\}$ și $Y_n = \{y_1, y_2, \dots, y_n\}$. Vom nota cu $c[i][j]$ lungimea celui mai lung subșir comun al șirurilor $X_i = \{x_1, x_2, \dots, x_i\}$ și $Y_j = \{y_1, y_2, \dots, y_j\}$. Dacă $x_m = y_n$, atunci această lungime este $1 + c[m-1][n-1]$, altfel ea este maximul dintre $c[m-1][n]$ și $c[m][n-1]$. Pe baza acestei observații, valorile $c[i][j]$ ($0 \leq i \leq m$, $0 \leq j \leq n$) se vor calcula sevențial, linie cu linie, de la stânga la dreapta:

```

ALGORITM_LCS ( $X_m = \{x_1, x_2, \dots, x_m\}$ ,  $Y_n = \{y_1, y_2, \dots, y_n\}$ )
  For ( $i = 0$ ;  $i < m$ ; step 1) Execute  $c[i][0] = 0$  End_For
  For ( $j = 0$ ;  $j < n$ ; step 1) Execute  $c[0][j] = 0$  End_For

  For ( $i = 1$ ;  $i < m$ ; step 1) Execute
    For ( $j = 1$ ;  $j < n$ ; step 1) Execute
      If ( $x_i = y_j$ ) Then
        Set  $c[i][j] = c[i-1][j-1] + 1$ 
      Else  $c[i][j] = max(c[i-1][j], c[i][j-1])$ 
      End_If
    End_For
  End_For

  return  $c[m][n]$ 
END_ALGORITM_LCS ( $X_m = \{x_1, x_2, \dots, x_m\}$ ,  $Y_n = \{y_1, y_2, \dots, y_n\}$ )

```

Pentru acest exemplu, tabelul din următoare este redat în figura următoare. Folosind semnele \downarrow , \leftarrow , \rightarrow pentru a specifica elementul vecin cu care s-a calculat o valoare $c[i][j]$ (sus, stânga, respectiv diagonală). Drumul pentru construcția subșirului maximal este marcat cu celele gri.

	<i>j</i>	0	1	2	3	4	5	6	7	8	9
<i>i</i>		<i>y_i</i>	90	7	10	70	71	81	30		
0	<i>x_i</i>	0	0	0	0	0	0	0	0	0	0
1	10	0	10	10	10	-1	-1	-1	-1	-1	-1
2	20	0	-1	-1	-1	11	11	11	11	11	11
3	20	0	11	11	11	11	11	11	11	11	11
4	10	0	11	11	11	-2	-2	-2	-2	-2	-2
5	40	0	11	11	11	12	12	12	12	12	12
6	20	0	11	11	11	12	12	-3	-3	-3	-3
7	30	0	11	11	11	12	12	13	13	13	-4
8	60	0	11	11	11	12	12	13	13	13	14

Exemplu pentru algoritmul LCS

După citirea celor două șiruri în vectorii *x*, respectiv *y*, vom declara dinamic matricea *c* cu *m* + 1 linii și *n* + 1 coloane, pe care o vom inițializa cu elemente 0:

```
vector< vector<int> > c(m+1, vector<int> (n+1, 0));
```

după care vom aplica algoritmul de mai sus pentru a calcula toate elementele sale iterativ. Pentru a refuza către încadrarea comună în vîrstă, vom folosi cînd încă vectorile comună se vor scrie în coloana *v* și la dreapta spre stînga. Se porneste cu linia *m* și coloana *n* și se parcurge drumul invers astfel: dacă valoarea *X* pentru linia curentă *ll* este egală cu valoarea *Y* pentru coloana curentă *cc*, atunci valoarea comună s-a obținut printr-o deplasare pe diagonală și ea se va adăuga lui *v*, după care ne deplasăm în coloana *cc* și în linia *ll* și continuăm procesul. Altfel se urmărește să se mutăm în coloană pentru care a fost executată o mișcare pe coloană, respectiv pe linie, atunci când a fost calculat elementul *c[ll][cc]*.

Program

```
#include <iostream>
#include <vector>

using std::vector;
using std::ifstream;
using std::ofstream;

int main()
```

```
ifstream in("lcs.in");
ofstream out("lcs.out");

int m, n;
vector<int> x, y;

if(in && !in.eof() && in>>m>>n){
    int i=0, t;
    for(; in && i<m && in>>t; ++i)
        x.push_back(t);
    for(i=0; in && i<n && in>>t; ++i)
        y.push_back(t);
}

vector< vector<int> > c(m+1, vector<int>(n+1, 0));

for(int i=1; i<=m; ++i)
    for(int j=1; j<=n; ++j){
        if(x[i-1]==y[j-1]){
            c[i][j] = c[i-1][j-1]+1;
            continue;
        }
        if (c[i-1][j]>=c[i][j-1]){
            c[i][j] = c[i-1][j];
        } else {
            c[i][j] = c[i][j-1];
        }
    }

int ll=m, cc=n;
vector<int> v;
while(ll || cc){
    if (ll && cc && x[ll-1]==y[cc-1]){
        v.push_back(x[ll-1]);
        --ll; --cc;
        continue;
    }
    if(ll && c[ll][cc]==c[ll-1][cc]){
        v.push_back(x[ll-1]);
        ll--;
    }
    if(cc && c[ll][cc]==c[ll][cc-1]){
        cc--;
    }
}

out << "Lungimea maxima: " << c[m][n] << endl;
vector<int>::reverse_iterator rit;
```

```

for(rIt=v.rbegin(); rIt != v.rend(); ++rIt){
    out << *rIt << " ";
}

return 0;
}

```

Probleme propuse

- Scrieți o metodă recursivă pentru listarea subșirului comun maximal.
- Generalizare.** Pentru un număr natural dat $k \geq 2$ și k siruri de numere întregi, determinați cel mai lung subșir maximal comun tuturor celor k siruri.

Problema 4. Triunghi de numere

Considerăm un triunghi de numere ca în exemplul de mai jos și drumurile care duc la numărul de pe prima linie în jos spre bază, astfel încât de fiecare dată se coboară pe linia următoare cu un număr la dreapta celui curent sau imediat dedesubtul acestuia. Găsiți un astfel de drum pentru care suma elementelor constituente să fie maximă.

Date de intrare. În fișierul *triunghi.in* se găsește la început numărul de linii ale triunghiului de numere dat, urmat de acesta, linie cu linie. Numărul maxim de linii ale triunghiului este 100 și fiecare număr este mai mic sau egal cu 100.

Date de ieșire. Scrieți pe prima linie a fișierului *triunghi.out* suma maximală găsită și pe următoarele linii pozițiile din triunghi ce descriu drumul unei soluții cu suma

<i>triunghi.in</i>	<i>triunghi.out</i>
7	Suma maxima = 346
10	1
82 81	2
.	3
2 14 35 7	3
41 3 52 26 15	3
32 90 11 87 56 23	4
54 65 89 32 71 9 31	5

(IOI, Schwerdtf. 1994)

Analiza problemei și proiectarea soluției

Alegerea la fiecare pas al celui mai mare dintre succesorii posibili pe linia următoare nu conduce de fiecare dată la soluția optimă (aceasta ar fi o soluție de tip *Greedy*). În exemplul nostru, dacă alegem de pe a doua linie a triunghiului 82 în loc de 81, atunci toate sumele care se pot obține mai departe sunt mai mici decât 346 (care este soluția optimală).

Conținutul triunghiului se va citi în tabloul $a[][]$ și construim tabloul bidimensional $s[][]$ cu semnificația:

- $s[i][j]$ este cea mai mare sumă până la linia i inclusiv, care conține elementul $a[i][j]$ ce se poate obține.

Pentru a salva drumul corespunzător lui $s[i][j]$, folosim tabloul $vPred[100][100]$, cu semnificația:

- $vPred[i][j] = -1$ pentru primul element;
- $vPred[i][j] = 0$, dacă suma optimală care se termină cu $a[i][j]$ îl conține și pe $a[i-1][j-1]$ (aceasta înseamnă că s-a executat ultima dată un pas spre dreapta-jos);
- $vPred[i][j] = 1$, dacă suma optimală care se termină cu $a[i][j]$ conține și pe $a[i-1][j]$ (aceasta înseamnă că s-a executat ultima dată un pas spre direct-jos);
- $vPred[i][0]$ este mereu 1, deoarece acest element poate fi ultimul al unei sume doar prin executarea unui pas direct în jos (nu există nici un predecesor) $vPred[i-1][-1]$).

Drumul este ulterior salvat în tabloul $w[]$, $w[i]$ este poziția numărului al i -lea pe acest drum optimal. După ce tablourile $s[][]$ și $vPred[]$ au fost construite, suma optimală se va găsi pe linia n a tabloului bidimensional $s[][]$. Mai întâi vom găsi această poziție:

```
w[n-1] = 0;
for(j=1; j<=n-1; j++)
    if(s[n-1][j] > s[n-1][w[n-1]]) w[n-1]= j;
```

Apoi vom utiliza $vPred[][]$ pentru a determina toate numerele necesare din triunghi:

```
for(i=n-2; i>=0; i--){
    if(vPred[i+1][w[i+1]] == 0) w[i] = w[i+1]-1; //pas dreapta-jos
    else w[i] = w[i+1]; //pas direct-jos
}
```

Program

```
#include <iostream>

void readData(int a[][100], int &n){
    int i, j;
    ifstream in("triunghi.in");
    in >> n;
    for(i=0; i<n; i++)
        for(j=0; j<=i; j++) in >> a[i][j];
}
```

```

void doProcess(int a[][100], int n, int s[][100],
               int vPred[][100], int w[]){
    int i, j;
    s[0][0] = a[0][0];
    vPred[0][0] = -1;
    for(i=1; i<n; i++){
        s[i][0] = a[i][0] + s[i-1][0];
        vPred[i][0] = 1;
        for(j=1; j<=i; j++){
            if(s[i-1][j-1] >= s[i-1][j]){
                vPred[i][j] = 0;
                s[i][j] = a[i][j]+s[i-1][j-1];
            }else{
                vPred[i][i] = 1;
                s[i][j] = a[i][j]+s[i-1][j];
            }
        }
    }
    w[n-1] = 0;
    for(j=0; j<=n-1; j++)
        if(s[n-1][j] > s[n-1][w[n-1]]) w[n-1] = j;
    for(i=n-2; i>=0; i--){
        if(vPred[i+1][w[i+1]] == 0) w[i] = w[i+1]-1;
        else w[i] = w[i+1];
    }
}

```

```

void writeData(int a[][100], int v[], int n){
    ofstream out("t1unghi.out");
    out << "s[" << maxima = << s[n-1][w[n-1]] << endl;
    for(int i=0; i<n; i++){
        out << w[i]+1 << endl;
    }
}

int main(){
    int i, j, n;
    int a[100][100], s[100][100], vPred[100][100];
    //...
    readData(a, n);
    doProcess(a, n, s, vPred, w);
    writeData(s, w, n);
    return 0;
}

```

$vPred[i][0]$ este mereu 1!

Suma optimă care se termină cu $s[i][0]$ va fi întotdeauna $s[i][0] + s[i-1][0]$.

Pentru o pereche (i, j) , $i, j \neq 0$ are loc:

- dacă $s[i-1][j-1] \geq s[i-1][j]$, atunci suma maximă cu ultim element $a[i][j]$ a fost obținută printr-un pas spre dreapta (cu ajutorul $s[i-1][j-1]$)

- altfel, suma maximă cu ultimul element $a[i][j]$ a fost obținută cu ajutorul unui pas direct-jos (cu elementul $a[i-1][j]$).

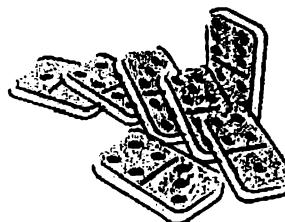
Pentru ambele cazuri $vPred[i][j]$ și $s[i][j]$ vor fi calculate corespunzător.

Probleme propuse

- Câte drumuri corecte sunt pentru un anumit n dat? Câte dintre acestea conțin o anumită poziție dată (i, j) ?
- Se poate să existe mai multe drumuri care conduc la obținerea sumei optime. Dezvoltați programul astfel încât să fie afișate toate aceste drumuri.
- Presupunem că atunci când se trece la linia următoare este permis și un pas spre stânga-jos. Modificați programul pentru problema astfel transformată.
- Pentru reprezentarea datelor am folosit tablourile bidimensionale $s[0][], s[0][]$ și $vPred[0][]$, dar de fapt se folosesc doar zonele de sub diagonala principală. Pentru fiecare astfel de tabelou, se alocă 100×100 elemente, când de fapt sunt necesare mult mai puține. Folosiți pentru stocarea dinamică a celor $\frac{n(n+1)}{2} = 1 + 2 + \dots + n$ elemente un `std::vector`. Modificați programul astfel încât să utilizați doar `std::vector` în locul tuturor tablourilor.

Problema 5. Domino

Se consideră un sir de piese de domino. Fiecare piesă poate fi rotită în jurul centrului ei cu 180° . Să se determine subșirul de piese de lungime maximă în care oricare două piese alăturăte au înscrise același număr: al doilea număr de pe prima piesă coincide cu primul număr înscris pe cea de-a două.



Într-o matrice de domino se pot să se găsească subșiruri de piese de domino, căreia pereche pe linie. Fiecare pereche conține numere între 0 și 6.

Date de ieșire. În fișierul `domino.out` trebuie afișat pe prima linie numărul maxim de piese al subșirului cu proprietatea din enunțul problemei, iar pe următoarele linii, căte una pe linie, trebuie scrise piesele în ordine. Exemplu:

domino.in	domino.out
5 6	6
1 1	6 5
2 5	5 2
3 2	2 3
6 4	5 5
5 1	5 2
5 3	
2 1	
1 4	
5 2	
3 3	

Analiza problemei și proiectarea soluției

Definim tipul *TPiece*, ce reprezintă o piesă și construim două tablouri:

- $v[200][2]$ cu semnificația:
 - $v[i][0]$ = lungimea maximă a unui subșir de piese care începe cu piesa i nerotită;
 - $v[i][1]$ = lungimea maximă a unui subșir de piese care începe cu piesa i rotită;
- $vNext[200][2]$, cu semnificația:
 - $vNext[i][0]$ = poziția succesorului piesei i în subșirul maximal care începe cu piesa i nerotită;
 - $vNext[i][1]$ = poziția succesorului piesei i în subșirul maximal care începe cu piesa i rotită.

Utilizăm variabilele:

- $iMax$ = indexul primei piese al subșirului maximal curent,
- bit (0 sau 1) = starea primei piese în subșirul maximal reprezentată de $iMax$ (0 – stare inițială – și 1 – rotită).

Metoda *doProcess()* prelucrează secvențial piesele de domino de la stânga spre dreapta și calculează valorile $v[i][0]$ și $v[i][1]$ pe baza numerelor deja calculate, după care se actualizează variabilele globale *iMax* și *bit*.

Metoda *write()* furnizează rezultatele și le scrie în fișierul de ieșire pe baza celor doi vectori de mai sus și a variabilelor globale *iMax* și *bit*. În variabila locală *aux* se va memora al doilea număr al piesei curente în subșirul maximal pentru a ști ce poziție are următoarea piesă (cunoaștem că ea dă niesă cu ajutorul vectorului *vNext[1][1]*).

Program

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

typedef pair<short, short> TPiece;

vector<TPiece> v;
int iMax, bit;
int vNext[200][2], v[200][2];

void read(){
    ifstream in("domino.in");
    TPiece p;
    while(in && !in.eof()){
        in >> p.first >> p.second;
        v.push_back(p);
    }
}
```

11 Programarea dinamică

```
    vP.push_back( p );
}

void doProcess(){
    int i, j, auxMax;
    int n = (int)vP.size();
    v[n-1][0] = 1; vNext[n-1][0] = n;
    v[n-1][1] = 1; vNext[n-1][1] = n;
    iMax = n-1; bit = 0;
    for(i=n-2; i>=0; i--){
        v[i][0] = 1; vNext[i][0] = n;
        v[i][1] = 1; vNext[i][1] = n;
        for(j=i+1; j<n; j++){
            if(vP[j].first==vP[i].second && v[i][0]<v[j][0]+1){
                v[i][0] = v[j][0]+1;
                vNext[i][0] = j;
            }
            if(vP[j].second==vP[i].second && v[i][0]<v[j][1]+1){
                v[i][0] = v[j][1]+1;
                vNext[i][0] = j;
            }
            if(vP[j].first==vP[i].first && v[i][1]<v[j][0]+1){
                v[i][1] = v[j][0]+1;
                vNext[i][1] = j;
            }
            if(vP[j].second==vP[i].first && v[i][1]<v[j][1]+1){
                v[i][1] = v[j][1]+1;
                vNext[i][1] = j;
            }
        }
        auxMax = v[i][0]>v[i][1]?v[i][0]:v[i][1];
        if(auxMax > v[iMax][bit]){
            bit = v[i][0]>v[i][1]?0:1;
        }
    }
}

void write(){
    int aux;
    ofstream out("domino.out");
    out << v[iMax][bit] << endl;
    for(i = 0; i < n; i++) vP[i].clear();
    if(bit == 1) {
        aux = vP[iMax].second;
        out << vP[iMax].first << " " << vP[iMax].second << endl;
    }
}
```

o alternativă pentru construcții de forma „if(A) then B else C”. Dacă A este diferit de zero sau de NULL, atunci rezultatul este B, altfel este C.

```

    }
else{
    aux = vP[iMax].first;
    out << vP[iMax].second << " " << vP[iMax].first << endl;
}
iMax = vNext[iMax][bit];
while(iMax != n){
    if(aux == vP[iMax].first){
        out << vP[iMax].first << " " << vP[iMax].second << endl;
        aux = vP[iMax].second;
        bit=0;
    }
    else{
        out << vP[iMax].second << " " << vP[iMax].first << endl;
        aux = vP[iMax].first;
        bit=1;
    }
    iMax = vNext[iMax][bit];
}
}

int main(){
    read();
    doProcess();
    write();
    return 0;
}

```

Probleme propuse

1. Modificați programul astfel încât să nu mai folosească variabilele globale.
2. Găsiți o alternativă mai scurtă pentru cele patru instrucțiuni de decizie *if*.
3. Rezolvați problema pentru cazul când se cere subsecvența maximală (piesele care sunt între două secvențe consecutive).

Problema 6. Împărțirea cadourilor

Doi frați au primit de Crăciun multe cadouri, care însă nu aveau pe ele etichete cu numele lor, și știau prețurile. Așa că, atunci când frații s-au trezit, au intrat într-o mare dilemă fiindcă nu știau cum să își împartă cadourile Moșului. Știind că fiecare cadou este o valoare cuprinsă între 1 și 100 lei, și că sunt maxim 50 de cadouri pentru ambii frați, scrieți un program care să împartă cât mai echitabil



cadourile, astfel încât sumele să fie cât mai apropiate.

Date de intrare. În fișierul *cadouri.in* se găsesc informațiile referitoare la cadouri: pe prima linie numărul total de cadouri, pe următoarea linie valorile lor.

Date de ieșire. În fișierul *cadouri.out* trebuie scrise două sume care sunt cele mai apropiate, corespunzătoare unei împărțiri a cadourilor, pe a doua linie valorile corespunzătoare cadourilor care însumează primul număr găsit, pe a treia linie, valorile corespunzătoare cadourilor care însumează al doilea număr găsit. Exemple:

<i>cadouri.in</i>	<i>cadouri.out</i>
7 28 7 11 8 9 7 27	48 49 28 11 9 7 8 7 27
15 12 43 8 90 13 5 78 34 1 97 31 65 80 15 17	294 295 12 43 90 13 5 34 97 8 78 1 31 65 80 15 17

(CEOI, Ungaria, 1995)

Analiza problemei și proiectarea soluției

O primă idee de rezolvare ar fi generarea tuturor submulțimilor de cadouri și selectarea celei mai convenabile. Această variantă este însă ineficientă din cauză că are o complexitate exponențială.

Cele n prețuri ale cadourilor vor fi salvate în tabloul *aPresent[]*. O altă variantă de rezolvare ar fi construirea unui vector *aS[]* cu semnificația:

- $aS[0] = 0$;
- $aS[j] = i_1 + i_2 + \dots + i_j$, unde i_1, i_2, \dots, i_j sunt j elemente ale submulțimii $\{i_1, i_2, \dots, i_n\}$ pentru a obține un total j este *aPresent[i]*;
- $aS[j] = 51$, atunci când suma j nu poate fi obținută.

Variabila *sum* conține suma tuturor cadourilor. Atunci când cadoul *aPresent[i]* va fi adăugat la *sum*, va fi verificat dacă suma totală este de 51 sau nu. Dacă este de 51, atunci *sum* - *aPresent[i]* (un cadou poate fi adăugat unei sume cel mult o singură dată și vom adăuga cadourile în ordine crescătoare). Prin sevență:

```

for(i=0; i<=5000; i++)
    aS[i] = 51;
aS[0] = 0;
for(i=1; i<=n; i++)
    for(j=0; j<=sum/2-aPresent[i]; j++)
        if(aS[j]<i && aS[j+aPresent[i]]==51)
            aS[j+aPresent[i]] = i;

```

se vor calcula valorile vectorului *aS[]*. În final, ne interesează să găsim cea mai apropiată sumă de mijloc; aceasta va fi una dintre sume și putem face afișarea celor

două sume optime. Pentru a reconstituî valorile celor două mulțimi de cadouri, vom scrie o funcție recursivă *recoverPresents()*. Ea va marca cu 1 în tabloul *a[]* cadourile considerate în prima mulțime, cu ajutorul căruia se construiește și a doua mulțime (cadourile rămase sunt marcate în *a[]* cu 0):

```
if(aS[i]>0){
    recoverPresents(i-aPresents[aS[i]], out);
    out << aPresents[aS[i]] << " ";
    a[aS[i]]=1;
}
```

Program

```
#include <iostream>

using namespace std;

short aS[5001];
short aPresents[51], a[51];
short n, i;
int j, sum=0;

void read(){
    ifstream in("cadouri.in");
    in >> n;
    for(i=1; i<=n; i++){
        in >> aPresents[i];
        if(aPresents[i]==0)
            a[i]=0;
        else
            a[i]=1;
    }
}

void recoverPresents(int i, ofstream& out){
    if(aS[i]>0){
        recoverPresents(i - aPresents[aS[i]], out);
        out << aPresents[aS[i]] << " ";
        a[aS[i]]=1;
    }
}

void write(){
    ofstream out("cadouri.out");
    for(j=sum/2; j>=0; j--)
        if(a[j]==1)
            out << j << " " << sum - j << endl;
    recoverPresents(j, out);
    out << endl;
```

11 Programarea dinamică

```
        break;
    }
    for(i=1; i<=n; i++)
        if(!a[i])
            out << aPresents[i] << " ";
}

int main(){
    read();
    for(i=0; i<=5000; i++) aS[i]=51;
    aS[0] = 0;
    for(i=1; i<=n; i++)
        for(j=0; j<= sum/2 - aPresents[i]; j++)
            if(aS[j]<i && 51==aS[j+aPresents[i]])
                aS[j+aPresents[i]] = i;
    write();
    return 0;
}
```

Probleme propuse

1. Modificați programul astfel încât să nu mai utilizeze variabile globale.
2. Modificați programul astfel încât în locul tuturor tablourilor să folosiți `std::vector`.
3. Reprezentați datele ca o listă cu perechi (*sumă, index_ultim_element_introdus*), care inițial e vidă și crește secvențial; odată cu procesarea unui nou element, se vor reactualiza/adăuga noi perechi la listă.

Problema 7. Sume asemănătoare

Fie un număr natural n și o mulțime M de numere naturale. Găsiți o submulțime U a lui M care îndeplinește condiția că ultimele trei cifre ale sumei elementelor sale sunt egale cu ultimele trei cifre ale sumei tuturor perechilor (i, j) din $M \times M$, unde $i < j$.
Pentru $n = 4$ și $M = \{65968, 65432, 65439876, 9078655, 56743, 54321, 6543298, 453129, 54321, 9990000, 567543\}$, U ar putea fi $\{9990000, 6543298, 56743, 65968\}$.

$$23980 + 675451 + 190223 + 78678 + 65892 + 45345 = 1079569$$

Pentru $n = 8$ și $M = \{65968, 65432, 65439876, 9078655, 56743, 54321, 6543298, 453129, 54321, 9990000, 567543\}$, U ar putea fi $\{9999999, 6543298, 56743, 65968\}$.

$$9999999 + 6543298 + 56743 + 65968 = 16666008$$

Date de intrare. Pe prima linie a fișierului *numere.in* se află numărul natural n , urmat de elementele mulțimii M (toate în intervalul $[0, 2000000000]$ și M are cel mult 5000 de elemente).

Date de ieșire. Scrieți în fișierul *numere.out* elementele mulțimii U care satisfac condiția din enunț. Dacă n are mai puțin de trei cifre atunci î se vor adăuga înainte numărul necesar de zerouri. Dacă nu există soluție, atunci scrieți în fișierul de ieșire: *nu există soluție!* Exemplu:

<i>numere.in</i>	<i>numere.out</i>
4569 45345 65892 78678 111153 190223 675451 543876 23980 99453 990121 656555 432908 12361	23980 675451 190223 78678 65892 45345
8 65968 65432 65439876 9078655 56743 6543298 453129 54321 9999999 567543	9999999 6543298 56743 65968
45678 65432 65439876 9078655 56743 54321 6543298 453129 54321 9999999 567543	nu există soluție!

Analiza problemei și proiectarea soluției

Problema este foarte asemănătoare cu problema anterioară 5.

Construim vectorul *vLast* cu 1000 de elemente, care sunt inițializate cu -1. Elementele vectorului sunt actualizate secvențial, pe măsură prelucrării elementelor mulțimii date. Semnificația elementelor sale este:

- *vLast[i]* – reprezintă indexul căruia să i se adauge la suma a trei cifre *xyz*;
- *vLast[i]* conține indexul i al elementului ultim adăugat (cu cel mai mare index) pentru a obține o sumă ale cărei ultime trei cifre sunt *xyz*.

Înlocuind, de exemplu, primul număr din mulțimea în teste cu 4569, valoarea *vLast[1]* se va seta pe 0 (indexul primului element al mulțimii).

Citim elementele mulțimii M în vectorul *vElem*. Elementele citite vor fi procesate iterativ ($aux \leftarrow vElem[i]$, pentru $i = 0, 1, 2, \dots$):

– calculăm $aux \% 1000$ și adăugăm rezultatul la suma a trei cifre *xyz*; – Până când

trei cifre ale lui *aux* ($vLast[aux \% 1000] == -1$), atunci salvăm indexul elementului i actual în *vLast[aux \% 1000]*:

```

    aux = vElem[i];
    aux = aux % 1000;
    if (vLast[aux % 1000] == -1)
        vLast[aux % 1000] = i;
}

```

- pentru toate sumele j deja calculate ($vLast[j] \neq -1$) calculăm posibilele seturi de trei cifre terminale ale sumelor pe care le putem obține prin adăugarea lui aux ($sum \leftarrow (aux + j) \% 1000$). Dacă valoarea corespunzătoare în $vLast$ este încă -1 , atunci ea se va actualiza cu i :

```
for(j=0; j<1000; j++){
    if(vLast[j]!=-1 && j!= aux%1000 && vLast[j]!=i){
        int sum = (aux+j)%1000;
        if(vLast[sum]==-1){
            vLast[sum] = i;
        }
    }
}
```

Dacă un element din vectorul $vLast$ a fost suprascris după inițializarea cu -1 , atunci el nu se va mai schimba. Cu ajutorul vectorului $vLast$ vom regăsi termenii sumei cerute:

```
while( vLast[i]>=0 ){
    out << " " << vElem[vLast[i]];
    j=i;
    if(i < vElem[vLast[i]]%1000){
        i = 1000+i-vElem[vLast[i]]%1000;
    }else {
        i = i-vElem[vLast[i]]%1000;
    }
    vLast[j]=-1;
}
```

Dacă $sum = \dots 127$ și ultimul termen al sumei $\dots 543$ ($127 > 543$), atunci $sum \leftarrow \dots 584$ ($1127 + 543$).

Dacă $sum = \dots 543$ și ultimul termen al sumei $\dots 127$ ($127 < 543$),

Programă

```
#include <iostream>
#include <vector>

int main(){
    vector<unsigned long long> vElem, v;
    vector<int> vLast;
    ...x;
    int start, i, j;
    ifstream in("numere.in");
    ofstream out("numere.out");
    in >> start;
```

```
while(in && !in.eof()){
    in >> aux;
    vElem.push_back(aux);
}

for(i=0; i<1000; i++)
    vLast.push_back(-1);

for(i=0; i<vElem.size(); i++){
    aux = vElem[i];
    if(-1==vLast[aux%1000]){
        vLast[aux%1000] = i;
    }
    for(j=0; j<1000; j++){
        if(vLast[j]!=-1 && j!= aux%1000 && vLast[j]!=i){
            int sum = (aux + j)%1000;
            if(-1==vLast[sum]){
                vLast[sum] = i;
            }
        }
    }
}

i=start;
if(i!=0) vLast[0] = -1;
if(vLast[i] != -1){

    cout << " ";
    cout << " " << vElem[vLast[i]];
    j=i;
    if(i<vElem[vLast[i]]%1000){
        i = 1000+i-vElem[vLast[i]]%1000;
    }else {
        i = 1000+i+vElem[vLast[i]]%1000;
    }
    vLast[j]=-1;
}

}

else out << "nu exista solutie!";
return 0;
}
```

11 Programarea dinamică

Probleme propuse

1. Câte posibilități de a obține o sumă cu trei cifre ce coincid cu cele ale lui n există? Dezvoltați programul astfel încât să scrie în fișierul de ieșire mai întâi numărul de sume posibile, urmat de una dintre ele:

numere1.in	numere1.out
4569 45345 65892 78678 111153 190223 675451 543876 23980 99453 990121 656555 432908 12361 65432 453219	...569: 7 23980 675451 190223 78678 65892 45345
781 65968 65432 65439876 9078655 56743 6543298 453129 54321 9999999 567543 4565 67543 45365 34546	...781: 5 567543 54321 6543298 56743 65439876

Subșirul indecșilor elementelor în submulțimea determinată este cel mai mic lexicografic relativ la toate posibilitățile de a obține o sumă corectă. Schimbați programul astfel încât să se scrie în fișierul de ieșire suma cu subșirul indecșilor cea mai mare lexicografic.

- Modificați programul astfel încât să se determine o sumă ce are număr maximal de termeni. Modificați apoi astfel încât să se scrie una cu număr minimal de termeni.
- Adăugăm și posibilitatea de a scădea termeni. Modificați programul corespunzător.
- Modificați programul astfel încât numerele date (inclusiv n) să poată fi și negative.
- Fie date n , k și elementele mulțimii M (toate din \mathbb{N} ; $0 \leq k \leq 1000$, celelalte mai mari sau egale cu 0 și mai mici sau egale cu 2.000.000.000). Găsiți o submulțime U a mulțimii date M cu proprietatea că ultimele trei cifre ale sumei puterilor de ordin
- Formulăm o nouă problemă în care elementele mulțimii M sunt numere reale pozitive cu maxim trei zecimale după virgulă. Ne interesează o submulțime a lui m cu proprietatea că suma elementelor sale să fie un număr natural. Scrieți un nou program care rezolvă problema. Urmăreșteți

numere2.in	numere2.out
0.123 6.789 10.125 115.117 11.03 12.12 7511.1 315.79 113.02 17.985	10.125 7511.1 315.79 17.985

- Este permisă și scăderea, iar numerele au voie să fie și negative. Modificați programul pentru aceste condiții în suplimentare.

Problema 8. Scoțieni la Oktoberfest



Într-o seară, un grup de scoțieni sunt în vizită la Oktoberfest și bineînțeles că vor să bea bere. Ei și-au procurat de ceva mai multă vreme bilete de intrare, care însă, din păcate, erau pentru două corturi diferite (au fost cumpărate prea târziu). Numărul de scoțieni este $2n$ și au pentru fiecare dintre cele două corturi câte n bilete, deci trebuie să se împartă în două grupe. Pentru a evita ceartă și discuțiile nesfărșite, au hotărât să dea cu banul. Ei au decis să arunce pe rând: cine aruncă pajură va merge în al doilea cort, altfel, va merge în primul cort. Normal că ei încețează cu aruncatul atunci când toate biletele pentru unul dintre corturi s-au epuizat. Ultimii doi la aruncat moneda sunt cei doi buni prieteni, Ian și Alistair, și ei își doresc foarte tare să meargă împreună în același cort, indiferent care. Ce probabilitate W există ca dorința prietenilor să se împlinească?

Întrebare. Pentru a calcula probabilitatea cerută, analizăm starea intermedieră după $2n - 2$ aruncări. Rămân în acest moment două bilete identice pentru unul din cele două corturi?

Date de intrare. În fișierul *oktoberfest.in* se găsesc mai multe numere naturale, fiecare cuprins între 1 și 500 și reprezentând pe n (este vorba despre $2n$ scoțieni).

Date de ieșire. Scrieți în fișierul *oktoberfest.out* pentru fiecare caz de intrare soluția în forma: :: W , unde n reprezintă o jumătate din numărul scoțienilor și probabilitatea W este scrisă cu trei zecimale exacte. Exemplu:

Input	Output
1	1: 0.000
2	2: 0.5000
3	3: 0.6250
4	4: 0.6875
5	5: 0.7312
6	6: 0.7539
56	56: 0.9241
345	345: 0.9696
432	432: 0.9728
500	500: 0.9747

Analiza problemei și proiectarea soluției

Într-o seară, un grup de scoțieni sunt în vizită la Oktoberfest și bineînțeles că vor să bea bere. Ei și-au procurat de ceva mai multă vreme bilete de intrare, care însă, din păcate, erau pentru două corturi diferite (au fost cumpărate prea târziu). Numărul de scoțieni este $2n$ și au pentru fiecare dintre cele două corturi câte n bilete, deci trebuie să se împartă în două grupe. Pentru a evita ceartă și discuțiile nesfărșite, au hotărât să dea cu banul. Ei au decis să arunce pe rând: cine aruncă pajură va merge în al doilea cort, altfel, va merge în primul cort. Normal că ei încețează cu aruncatul atunci când toate biletele pentru unul dintre corturi s-au epuizat. Ultimii doi la aruncat moneda sunt cei doi buni prieteni, Ian și Alistair, și ei își doresc foarte tare să meargă împreună în același cort, indiferent care. Ce probabilitate W există ca dorința prietenilor să se împlinească? Însă determinarea probabilității p ca Ian și Alistair să nu ajungă în același cort. Probabilitatea ca ei să meargă în același cort va fi $1-p$. Notăm cu $P(m, s)$

probabilitatea ca din m aruncări ale monedei să apară de s ori cap. Pentru cazurile $s = 0$ și $s = m$, probabilitatea $P(m, s)$ este $\frac{1}{2^m}$:

$$P(m, 0) = P(m, m) = \frac{1}{2^m}$$

Explicația este că probabilitatea ca de la prima aruncare să apară cap este $\frac{1}{2}$, probabilitatea să apară cap și din a doua aruncare este $\frac{1}{2} \cdot \frac{1}{2}$, iar de a apărea și în a treia este $\frac{1}{4} \cdot \frac{1}{2} = \frac{1}{2^3}$. Această probabilitate este egală cu probabilitatea ca dintre toți vectorii de lungime m cu elemente 0 și 1 să îl alegem pe acela care conține doar zerouri, respectiv doar elemente 1 (există numai un astfel de vector și numărul total al vectorilor este 2^m). Dacă $s \neq 0$ și $s \neq m$, atunci are loc formula recursivă:

$$P(m, s) = \frac{1}{2}P(m-1, s) + \frac{1}{2}P(m-1, s-1)$$

Posibilitatea ca din m aruncări să obținem de s ori cap este	$P(m, s) =$
să se arunce pajură după ce din $m-1$ aruncări a fost de s ori cap	$\frac{1}{2}P(m-1, s)$
sau	$+$
șă se arunce cap după ce din $m-1$ aruncări a fost de $(s-1)$ ori cap	$\frac{1}{2}P(m-1, s-1)$

Așadar:

$$P(m, s) = \begin{cases} \frac{1}{2^m}, & \text{dacă } s \in \{0, m\} \\ \frac{1}{2} \cdot P(m-1, s) + \frac{1}{2} \cdot P(m-1, s-1), & \text{dacă } s \in \{1, 2, \dots, m-1\} \end{cases}$$

Avem astfel o formulă recursivă și putem calcula probabilitatea $P(2n-2, n-1)$ (ca din $2n-2$ aruncări să se arunce de $n-1$ ori cap și de $n-1$ ori pajură, ceea ce înseamnă că prietenii vor fi separați). Răspunsul la problemă va fi deci $1 - P(2n-2, n-1)$ (ei nu vor fi separați).

Program 1. O primă variantă recursivă:

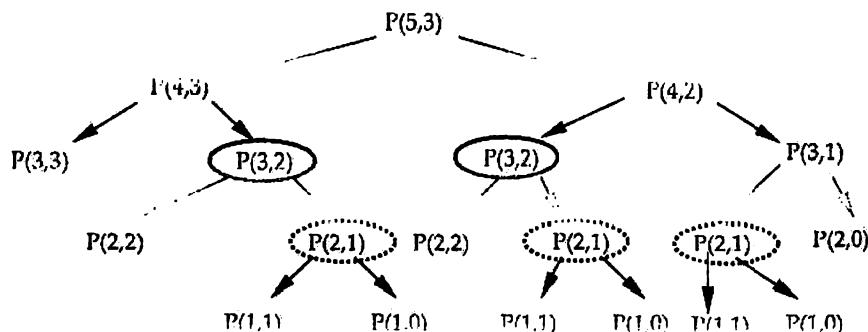
```
#include <iostream>

double f(int n){
    double r=1;
    for(int i=0; i<n; i++) r*=0.5;
    return r;
}

double P(int m, int q){
    if(q==0 || q==m) return f(m);
    return 0.5*P(m-1, q) + 0.5*P(m-1, q-1);
}

int main(){
    int n;
    std::cout << "Numar perechi: ";
    std::cin >> n;
    std::cout << "Probabilitatea> ";
    std::cout.precision( 4 ); std::cout << 1 - P(2*n-2, n-1);
    return 0;
}
```

Această variantă are însă un mare neajuns: timpul de execuție! Vom vedea că timpul de execuție pentru $n > 12$ crește considerabil. Aceasta se întâmplă din cauza faptului că aceleași valori (ca și în cazul sirului lui Fibonacci) se calculează de mai multe ori (adică, aceeași rezolvare este folosită). Acest lucru este posibil $P(5, 3)$:



Acest algoritm arată încă o dată că folosirea unei metode recursive nu este întotdeauna cea mai bună alegeră.

Deoarece numărul n al perechilor poate crește până la 500, este evident că trebuie să găsim o altă metodă de rezolvare. Putem să construim un tablou bidimensional $p[0..m][0..m]$, cu semnificația că $p[i][j]$ este probabilitatea ca din i aruncări să se obțină de j ori cap.

```

p[0][0]
p[1][0]  p[1][1]
p[2][0]  p[2][1]  p[2][2]
p[3][0]  p[3][1]  p[3][2]  p[3][3]
.....
p[m][0]  p[m][1]  p[m][2]  .....p[m][m]

```

Program 2

```

#include <iostream>

const int NMAX = 500;
using namespace std;

int main(){
    int n, i, j;
    float p[2*NMAX+1][2*NMAX+1];
    ifstream fin("oktoberfest.in");
    ofstream fout("oktoberfest.out");
    p[0][0] = 1;
    .....
    for(i=2; i<=2*NMAX-2; i++){
        p[i][0] = p[i][i] = p[i-1][0]*0.5;
        for(j=1; j<i; j++)
            p[i][j]=0.5*p[i-1][j]+0.5*p[i-1][j-1];
    }
    .....
    fin >> n;
    fout.width(3);
    fout << n << ":";
    fout.precision(4);
    cout << "Linea " << n-1 << endl;
    fout << 1-p[2*n-2][n-1] << endl;
}
return 0;
}

```

La început, se calculează valorile de sub diagonala principală a tabloului $p[][]$, linie cu linie, astfel încât pentru determinarea valorilor dintr-o linie se folosesc cele din linia anterioară. Această metodă nu mai are probleme cu timpul de execuție, însă are o altă

deficiență majoră: spațiul folosit. Pentru $NMAX = 500$, tabloul $p[][]$ are nevoie de 1002101 de tip *float*. De obicei, un element de tip *float* are 4 octeți, ceea ce înseamnă că avem nevoie de $4 \cdot 1002001 = 4008004$ octeți (aproximativ 3,9 MB). La execuția programului, putem obține o eroare de genul: „*Unhandled exception at 0x00435415 in oktoberfest.exe: 0xC00000FD: Stack overflow*”.

Pentru a evita și acest lucru, putem folosi doi vectori (unul pentru valorile din linia predecesoare și unul pentru cea actuală), care vor fi mereu actualizați, și un vector *c*, care conține valorile $P(i, i/2)$ pentru i număr par. Mai mult, putem folosi doar doi vectori, unul cu probabilitățile $P(i, i/2)$ și un altul ajutător, care conține mereu probabilitățile $P(i, 0)$, $P(i, 1)$ până la $P(i, i)$ pentru linia i curentă. Astfel, obținem următoarea variantă:

Program 3

```
#include <fstream>
#include <vector>

const int NMAX = 500;
using namespace std;

int main(){
    int n, i, j;
    float aux1, aux2;
    vector<float> a, c;
    ifstream fin("oktoberfest.in");
    if(!fin)
        cout << "fin不成";
    a.push_back(0.5);
    c.push_back(1);
    for(i=2; i<=1000; i++){
        aux1 = a[0];
        for(j=1; j<i; j++){
            aux2 = a[j];
            a[j] = (float)(0.5*aux1 + 0.5*a[j]);
            aux1=aux2;
        }
        a.push_back(a[0]);
        if(0==i%2) c.push_back(a[i/2]);
    }
    cout << "n = " << n << endl();
    fin >> n;
    fout.width(3);
    fout << n << " perechi: ";
    fout.precision(4);
```

```

    fout.flags(ios::fixed);
    fout << 1-c[n-1] << endl;
}
return 0;
}

```

Altă metodă de rezolvare

Observăm că formula pentru $P(m, q)$ este foarte asemănătoare cu formula recursivă pentru combinații (coeficienții binomiali). Pentru rezolvarea problemei, avem de fapt nevoie doar de valorile $P(2n, n)$, pentru $n \in \{0, 1, 2, \dots, 500\}$. Notăm $P(2n, n)$ cu T_n și considerăm următoarele mulțimi:

- M_1 = mulțimea tuturor vectorilor de lungime $2n$, care conțin doar 0 și 1 (0 și 1 au semnificația de cap, respectiv pajură);
- M_2 = mulțimea vectorilor de $2n$ care conțin n elemente 0 și n elemente 1, cu condiția ca valorile ultimelor două poziții să fie diferite.

Cu această notație, T_n este $\frac{|M_2|}{|M_1|}$.

$$|M_1| \text{ este } 2^{2n}, |M_2| \text{ este jumătate din } C_{2n}^n, \text{ de unde rezultă: } T_n = \frac{C_{2n}^n}{2^{2n+1}}. \quad (1)$$

Din (1) rezultă reprezentarea recursivă pentru sirul T_n :

$$T_n = \frac{1}{2n} \cdot T_{n-1} \cdot (2n-1) \quad (n > 0, \text{ și } T_0 = 1) \quad (2)$$

Pe baza formulei (2), scriem un nou program:

```

#include <iostream>
#include <vector>

const int NMAX = 500;
std::vector<float> T;

int main(){
    int n, i, j;
    float aux1, aux2;
    vector<int> c(1);
    ifstream fin("oktoberfest.in");
    ofstream fout("oktoberfest.out");
    T.push_back(1);

```

$$T_n = \frac{C_{2n}^n}{2^{2n+1}}$$

$$\begin{aligned} T[0] &\leftarrow 1 \\ T[i] &\leftarrow \frac{2i-1}{2i} T[i-1] \end{aligned}$$

```

for(i=1; i<=NMAX; i++){
    T.push_back((float)((2*i-1.0)/(2*i)*T[i-1]));
}

while(fin && !fin.eof()){
    fin >> n;
    fout.width(3);
    fout << n << ":" ;
    fout.precision(4);
    fout.flags(ios::fixed);
    fout << 1-T[n-1] << endl;
}
return 0;
}

```

Probleme propuse

- Cât de mare este probabilitatea ca prietenii să ajungă amândoi în cortul 1? Cât de mare este probabilitatea ca Ian să meargă în cortul 1 și Alistair în cortul 2?
- În programul 1 folosim funcția $f()$ pentru calcularea valorilor $1/2^n$. De ce următoarea variantă nu este corectă?

```

double f(int n){
    long int p = 1;
    for(int i=0; i<n; i++) p*=2;
    return 1/p;
}

```

- În al doilea tablou, declarăm un tablou bidimensional $p[][]$, dar este folosit doar domeniul de sub diagonala principală. Optimizați folosirea spațiului cu ajutorul unei reprezentări ca vector numai a acestei zone.
- C_2^n este numărul tuturor combinațiilor de a obține n pe $2n$.
- Explicați de ce $|M_2|$ este o jumătate din C_2^n .
- Găsiți o relație dintre T_n și numerele lui Catalan.

Problema 9. Calul pe tablă de sah

O problemă cunoscută este determinarea posibilităților ca un cal să parcurgă o tablă de șah trecând prin fiecare câmp exact o singură dată (problemă rezolvată în capitolele *Greedy* și *Iterativă*). Vă vom considera în acest capitol o problemă similară: se dau cele două poziții de start, respectiv de final și dimensiunea n a tablei de șah și se cere cel mai scurt drum al unui cal între cele



două câmpuri date.

Date de intrare. În fișierul `cal.in` se găsesc cinci numere naturale n , l_1 , c_1 , l_2 , c_2 , cu semnificațiiile: n este numărul de linii și coloane al tablei de săh ($4 \leq n \leq 20$) și l_1, c_1, l_2, c_2 sunt „coordonatele” câmpurilor de start și de sfârșit.

Date de ieșire. Scrieți în fișierul de ieșire `cal.out` mai întâi lungimea celui mai scurt drum și apoi un astfel de drum de lungime minimă (dacă există mai multe, atunci se va scrie doar unul singur). Exemplu:

cal.in	cal.out
8 1 1 8 7	Cel mai scurt drum: 5 (1, 1)(3, 2)(5, 3)(7, 4)(6, 6)(8, 7)
5 5 2 1 4	Cel mai scurt drum: 2 (5, 2)(3, 3)(1, 4)
15 2 13 14 15	Cel mai scurt drum: 6 (2, 13)(4, 14)(6, 15)(8, 14)(10, 15)(12, 14)(14, 15)

Analiza problemei și proiectarea soluției

Definim *distanța* (relativ la două poziții date) ca fiind numărul minim de mutări ale calului pentru a ajunge de la una la alta. Vom construi o tabelă ce reprezintă tabla de săh și fiecare celulă conține distanța de la poziția de start la câmpul respectiv. Pentru al doilea exemplu din fișierul de intrare, această tabelă va fi:

3	2	3	2	3
7	1	2	3	4
5	2	1	4	3
2	3	2	1	2
3	0	3	2	3

Câmpul încercuit este câmpul de start în exemplul din fișierul de intrare. În ceea ce urmărește claritatea algoritmului în program vom folosi variabilele globale:

```
int Table[NMAX][NMAX]; // Tabela distanțelor
int n; // Numărul de linii și coloane ale tablei
int l1, c1, l2, c2; // Poziții în predefinită, cu reprezentările
```

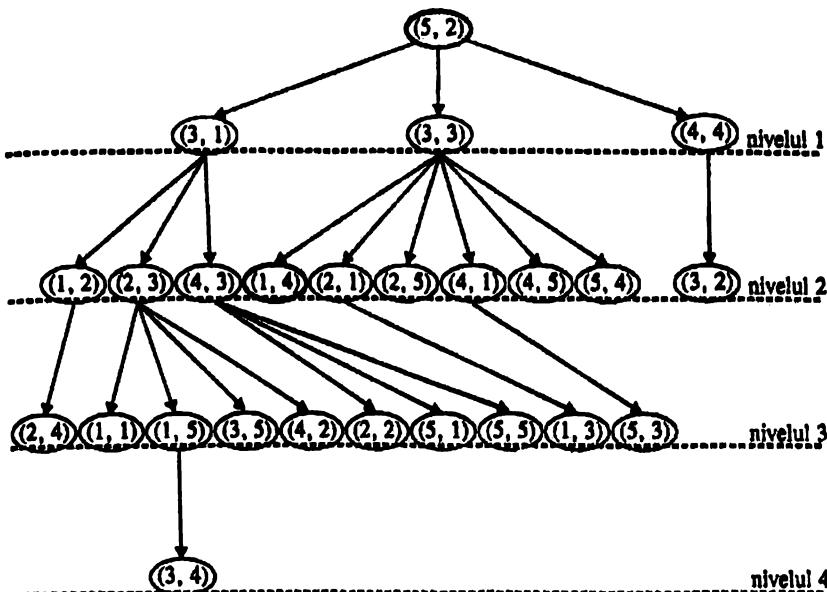
Vom scrie o metodă recursivă `doMove()` care parurge pătratele tablei începând cu cel inițial în ordinea dată de pasul calului. În implementarea acestei funcții, vom

- `void initTable()`: inițializează fiecare celulă a tablei cu `INT_MAX` (un fel de plus infinit); atunci când celul ajunge într-un anumit câmp, vom salva distanța de la câmpul inițial până aici, care sigur va fi mai mică decât `INT_MAX`;
- `bool onTheTable()`: returnează `true` dacă punctul (i, j) se află pe tablă, altfel returnează `false`;
- `void recoverWay()`: metodă recursivă care construiește drumul minim. Cele $n \times n$ câmpuri ale tablei sunt salvate într-un vector; de exemplu, pentru $n = 5$:

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

- câmpul (i, j) va fi salvat în vector la poziția $i \cdot n + j$;
- pentru un element k din vector, câmpul în tabel este $(k \text{ div } n, k \text{ mod } n)$;
- liniile și coloanele vor fi numerotate intern de la 0; de aceea, scriem în fișierul de ieșire valorile $k \text{ div } n + 1$ și $k \text{ mod } n + 1$ (în descrierea problemei se începe de la 1!);
- `void doMove()`: metodă recursivă ce are ca parametru un câmp al tablei pentru care distanța a fost deja calculată. Corespunzător acestelui poziții, se calculează toate câmpurile încă neatinse și care pot fi direct succesoare ale câmpului dat ca parametru și aplecă recursiv metoda pentru aceste câmpuri (procesul este finit, singură dată).

O altă modalitate de rezolvare ar fi folosirea unui arbore. Nodurile sale reprezintă câmpurile tablei de șah, iar rădăcina este câmpul inițial. Vom introduce căte o mulțime de noduri în arbore, ceea ce înseamnă că există o mulțime de căi între un câmpul (i, j) și câmpul (p, q) . Nivelurile arborelui pot fi parcuse folosind algoritmul de parcurgere în lățime (BFS – Breadth First Search), până când se ajunge la nodul final. Pentru exemplul nostru:



Program

```

#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;
const int NMAX = 20;
typedef pair<int, int> TIntPair;
vector<TIntPair> Table[NMAX];
int n;
int xNew, yNew;
bool onTheTable(int x, int y);

void initTable(){
    for(int i=0; i<n; i++)
        for(int j=0; j<n; j++)
            Table[i].push_back({j, i});
}

bool onTheTable(int x, int y){
    return (

```

```

        x>=0 && x<n &&
        y>=0 && y<n);

}

void doMove(int x, int y, int c){
    int i, j;
    TIntVector v;
    TIntPair p;
    for(i=-2; i<3; i++)
        for(j=-2; j<3; j++)
            if(2 == abs(i*j)){
                xNew = x + i;
                yNew = y + j;
                if(onTheTable(xNew, yNew) && Table[xNew][yNew]>c){
                    Pred[xNew*n + yNew] = x*n + y;
                    Table[xNew][yNew] = c;
                    v.push_back(TIntPair(xNew, yNew));
                }
            }
    while(v.size()){
        p = v.back();
        doMove(p.first, p.second, c+1);
        v.pop_back();
    }
}

void recoverWay(TIntVector &out, int i, int c)
{
    int k;
    if(c){
        k = Pred[i];
        recoverWay(out, k, c-1);
        out << "(" << k/n + 1 << ", ";
    }
}

int main(){
    int n, l1, l2, c1, c2;
    ifstream in("cal.in");
    ofstream out("cal.out");
    in >> n >> l1 >> c1 >> l2 >> c2;
    initTable();
    l1++; l2++; c1++; c2++;
    doMove(l1, c1, 1);
    if( Table[l2][c2] == INT_MAX ){
        out << "nu exista solutie";
        return 0;
    }
}

```

```

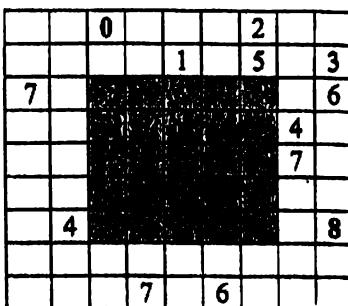
    }
    out << "Cel mai scurt drum: "
    << Table[12][c2] << endl;
recoverWay(out, 12*n+c2, Table[12][c2]+1);
return 0;
}

```

Probleme propuse

1. Găsiți cel mai scurt drum al calului atunci când acesta are voie doar pe cele două rânduri exterioare ale tablei (câmpurile gri sunt închise). Dacă nu există soluție, atunci scrieți în fișierul de ieșire: *nu există soluție*.

			4		
		2			
6			3		
			1		
0					

Două exemple ($n = 5$ și $n = 9$)

5 5 2 1 4	Cel mai scurt drum: 4 (5, 2)(4, 4)(2, 3)(3, 5)(1, 4)
9 1 3 7 9	Cel mai scurt drum: 8 (1, 3)(2, 5)(1, 7)(2, 9)(4, 8)(2, 7)(3, 9) (5, 8)(7, 9)
15 14 13 1 2	Cel mai scurt drum: 12 (14, 13)(13, 15)(11, 14)(9, 15)(7, 14)(5, 15) (3, 14)(2, 12)(1, 10)(2, 8)(1, 6)(2, 4)(1, 2)

Unde în acel loc este problema? Căci nu e unul cel puțin să se opereze totă această zonă exterioară a tablei (două rânduri), astfel încât fiecare câmp este pașit doar o singură dată (există numai $8n - 16$ câmpuri permise).

2. Scrieți un program care implementează soluția alternativă cu reprezentarea ca arbore și parcurgerea în lățime.
3. Pentru a găsi toate pozițiile succesoare ale unui câmp dat, folosim metoda *doMove()*. Se dă $n = 5$ și câmpurile $(1, 4)$, $(2, 5)$ și $(3, 3)$. Scrieți pe hârtie noile posibile poziții pentru fiecare dintre câmpurile date cu ajutorul buclelor *for*:

```

for(i=-2; i<3; i++)
  for(j=-2; j<3; j++)
    if(2==abs(i*j)) {
      xNew = x + i;
      yNew = y + j;
    }
  
```

4. Numărul cailor. Căți cai se pot amplasa pe o tablă de șah specială de dimensiune $m \times n$ ($1 \leq m \leq 20, 1 \leq n \leq 20$), astfel încât oricare doi să nu se atace între ei?

Problema 10. Fluctuații la bursă

Utilizarea calculatoarelor în industria financiară se extinde și în calculul bursier, pentru a exploata avantajele unor mici fluctuații de preț a mai multor valute. *Arbitrajul* este schimbarea banilor dintr-o valută în alta în scopul obținerii de profit. De exemplu, dacă schimbăm 1,00\$ în lire sterline, vom cumpăra 0,7 lire, cu o liră vom cumpăra 9,5 franci francezi și cu un franc francez vom cumpăra 0,16\$. Atunci, vom obține $1 \times 0,7 \times 9,5 \times 0,16 = 1,064 \$$, ceea ce înseamnă un profit de 6,4%. Trebuie să scrieți un program care determină dacă există o secvență care să aducă profit, asemănătoare cu cea descrisă mai sus. Pentru a fi o secvență corectă, aceasta ar trebui să înceapă și să se sfârsească cu aceeași valută, putând fi considerată oricare dintre ele. *Date de intrare.* Fișierul de intrare *bursa.in* conține una sau mai multe table de conversie. Trebuie să rezolvați problema arbitrajului pentru fiecare tabelă în parte. Fiecare tabelă este precedată de un număr întreg n ($2 \leq n \leq 20$) pe o linie care este cînd următoarea linie, urmat de elementele tablăi, cîndcînță ea încearcă să se diagonala principală lipsesc (presupunem că au toate valoarea 1). Prinul rând al tabliei reprezintă ratele de conversie ale țării 1 relativ la celelalte $n - 1$ țări, adică elementul i reprezintă valoarea cu care poate fi obținută o unitate din valută țării i . Dacă fiecare tabelă este definită de $n + 1$ linii: una pentru n – numărul de valute – și n linii reprezentând tabela de conversie.

Date de ieșire. Fișierul de ieșire este *bursa.out*. Pentru fiecare tabelă din fișierul de intrare trebuie să determinați dacă există o secvență astfel încât profitul să fie mai mare decât un procent (0,01). Dacă o astfel de secvență există, trebuie să afișați o secvență de lungime minimă, adică una dintre secvențele care folosesc cel mai mic număr de valute pentru a obține profitul. Dacă nu există o secvență de cel mult lungimea n , trebuie să se vor scrie în fișierul de ieșire „0”. Dacă există și mai de o secvență care va furniza un profit mai mare decât 1%, atunci va trebui să afișați o secvență de lungime minimă, adică una dintre secvențele care folosesc cel mai mic număr de valute pentru a obține profitul. Dacă nu există o secvență de cel mult lungimea n , trebuie să se vor scrie în fișierul de ieșire „0”. Atunci secvența de ieșire va fi formată din numărul de secvențe de lungimea n și în fișierul de ieșire cazurile se vor numerota, iar formatul va trebui să fie ca în exemplul de mai jos. Exemplu:

bursa.in	bursa.out
<pre> 3 1.2 0.89 0.88 5.1 1.1 0.15 4 3.1 0.0023 0.35 0.21 0.00353 8.13 200 180.559 10.339 2.11 0.089 0.06111 2 2.0 0.45 </pre>	<pre> Caz 1: · Valuta 2 2 -> 1 -> 2 1.056 Caz 2: · Valuta 3 3 -> 1 -> 2 -> 3 2.189 Caz 3: Nu există secvență de arbitraj! </pre>

(ACM Internet Programming Contest, 1990)

Analiza problemei și proiectarea soluției

Vom rezolva problema adaptând un algoritm de tip Roy-Warshall asupra matricei $tabelă[i][j]$, care conține inițial tabela de schimb. Pentru a regăsi drumul parcurs în scopul obținerii unei secvențe de arbitraj, vom utiliza un vector bidimensional $pred[i][j]$ cu semnificația: $pred[i][j]$ este predecesorul lui i pe drumul de valoare maximă dintre i și j (valoarea maximă care se poate obține schimbând valută i în valută j).

- *bool read()* va citi un set de date din fișierul de intrare; dacă mai există un astfel de set, se va returna 1, altfel se va returna 0. În cadrul acestei funcții, se inițializează și vectorul $pred[i][j] = j$ (în acest moment, predecesorul lui j pe drumul de la i la j este j și valoarea este 0).
- *bool gaseste()* conține un algoritm Roy-Warshall modificat; întrucât prima iterație se referă la numărul nodurilor intermediare dintr-un drum, deducem că prima dată când vom obține o valoare pe diagonala principală $\geq 1,01$, putem să ne oprim și să scriem rezultatele (nu ne interesează decât un răspuns),

... și să nu urmărim să obțină valută, pe baza vectorului $pred[]$.

Program

```

#include <iostream.h>
#define DIM_MAX 21

int n, nCaz = 0;
float tabelă[DIM_MAX][DIM_MAX];
int pred[DIM_MAX][DIM_MAX];

```

```

bool read(std::ifstream& in){
    if(in && !in.eof() && in>>n){
        for(int i=0; i<n; i++){
            tabla[i][i]=1.0;
            for(int j=0; j<n; j++)
                if(i != j){
                    in >> tabla[i][j];
                    pred[i][j]=i;
                }
            pred[i][i] = -1;
        }
    } else return false;
    return true;
}

void reconstituieDrum(int i, int j, std::ofstream& out){
    int t = pred[i][j];
    if(i!=t){
        reconstituieDrum(i, t, out);
        out << " -> " << t + 1;
    }
}

bool gaseste(std::ofstream& out){
    int x, y, j;
    for(y=0; y<n; y++)
        for(x=0; x<n; x++)
            if(tabla[x][y]<tabla[x][j]){
                pred[x][j] = pred[y][j];
                tabla[x][j] = tabla[x][y] * tabla[y][j];
                if (x==y & tabla[x][j]>=1.01){
                    out << "\n Valuta " << x+1
                        << endl << " -> " << y+1;
                    reconstituieDrum(x, x, out);
                    out << " -> " << x+1 << std::endl;
                    out.precision(3); //out.width(9);
                    out << tabla[x][x];
                    rotire();
                }
            }
    return false;
}

int main(){
    std::ifstream in("bursa.in");
    std::ofstream out("bursa.out");
    while(read(in)){

```

11 Programarea dinamică

3

```
    out << "\nCaz " << ++nCaz << ": ";
    bool ok = gaseste(out);
    if(!ok)
        out << "\n Nu exista secventa de arbitraj!\n";
}
return 0;
}
```

Probleme propuse

1. Modificați programul de mai sus astfel încât să se afișeze o secvență de arbitraj care aduce profit maxim (nu este nevoie neapărat ca lungimea să fie maximă).
2. Modificați programul de mai sus astfel încât să se afișeze toate secvențele de arbitraj care aduc un profit de peste 1%, de lungime minimă.

Problema 11. Sume de produse

Se dă numerele naturale $n < 30$ și a_1, a_2, \dots, a_n , fiecare mai mic decât 101. Calculați numerele b_1, b_2, \dots, b_n , astfel încât:

$$b_i = \sum_{1 \leq k_1 < k_2 < \dots < k_i \leq n} a_{k_1} \cdot a_{k_2} \cdots \cdot a_{k_i}$$

De exemplu, pentru $n = 3$:

$$b_1 = a_1$$

$$b_2 = a_1 \cdot a_2 + a_1 \cdot a_3 + a_2 \cdot a_3$$

$$b_3 = a_1 \cdot a_2 \cdot a_3$$

Valorile b_1, b_2, \dots, b_n „încap” în tipul *unsigned long long*.

Date de intrare. În fișierul *prodsum.in* se pășesc numerele a_1, a_2, \dots, a_n .

Date de ieșire. Scrieți numerele b_1, b_2, \dots, b_n în fișierul *prodsum.out* (câte unul pe linie) și înaintea fiecărui precizați suma cifrelor sale, ca în exemplu:

prodsum.in	prodsum.out
2	8 35
3	5 50
4	6 24
2	8 17
1	6 105
3	14 374
4	15 168

(ACM, South-Eastern European Regionals, 1999, enunț modificat)

Analiza problemei și proiectarea soluției

Vom folosi relațiile lui Viète.

Dacă x_1, x_2, \dots, x_n sunt rădăcinile ecuației:

$$x^n + c_{n-1}x^{n-1} + c_{n-2}x^{n-2} + \dots + c_0 = 0 \quad (1)$$

atunci au loc:

$$x_1 + x_2 + \dots + x_n = \sum_{i=1}^n x_i = -c_{n-1}$$

$$x_1x_2 + x_1x_3 + \dots + x_{n-1}x_n = \sum_{\substack{i,j=1 \\ i < j}}^n x_i x_j = c_{n-2}$$

$$x_1x_2x_3 + x_1x_2x_4 + \dots + x_{n-2}x_{n-1}x_n = \sum_{\substack{i,j,k=1 \\ i < j < k}}^n x_i x_j x_k = -c_{n-3} \quad (2)$$

$$\dots$$

$$x_1x_2 \cdots x_n = (-1)^n c_0$$

Ecuația (1) poate fi scrisă și sub forma:

$$(x - x_1)(x - x_2) \cdots (x - x_n) = 0 \quad (3)$$

Dacă presupunem că numerele date a_1, a_2, \dots, a_n sunt rădăcinile ecuației (3), atunci rezultă, pe baza relațiilor lui Viète, că numerele cerute sunt coeficienții ecuației (1)

Coefficienții polinomului (1) se vor calcula pas cu pas și $P[i] = a_i$ pentru orice $i = 0, 1, \dots, n$. La pasul al i -lea se va multiplica polinomul actual cu polinomul $(X - a_i)$. Polinomul de start este $X - a_0$, de aceea inițializăm:

- $P[0] = -c[0]$;

- $c[0] = 1$.

Pentru toți k de la 1 la $n-1$ considerăm următoarea transformare:

$$P[k] \leftarrow P[k] * X + c[k], \quad c[k] \leftarrow c[k] * a[k]$$

Noii coeficienții ai lui $P[]$ sunt coeficienții calculați simultan pe baza înmulțirii (4):

- $P[0] \leftarrow -P[0] * a[k];$

- $P[j] \leftarrow P[j-1] - P[j] * a[k]$, pentru orice j de la 1 la k ;
- $P[k+1] \leftarrow P[k]$.

Dacă schimbăm semnele conform formulelor lui Viète, atunci numerele cerute b_1, b_2, \dots, b_n sunt $(-1)^n P[n], (-1)^{n-1} P[n-1], \dots, P[0]$. Ele se vor copia în tabloul $a[]$:

```
j = 1;
if(n%2) j = -1;
for(k=0; k<=n; k++) {a[k] = j*p[k]; j *= -1;}
```

În final, calculăm conform enunțului și suma cifrelor pentru fiecare astfel de număr b_i , folosind funcția *sumDigits()*.

Program

```
#include <iostream>
#include <vector>

using namespace std;

void read(unsigned long long a[31], short& n){
    ifstream in("prodsum.in");
    n = 0;
    while(in && !in.eof()){
        in >> a[n];
    }
}

void process(unsigned long long a[31], short n){
    short j, k;
    unsigned long long p[31] = {0};
    if(j > n) return;
    p[0] = -a[0];
    p[1] = 1;
    for(k=1; k<n; k++){
        p[0] = (-1) * p[0] + a[k];
        for(j=k+1; j<n; j++)
            q[j] = p[j-1] - p[j] * a[k];
    }
    q[k+1] = p[k];
    for(i=0; i<=k+1; i++) r[i] = q[i];
}

j = 1;
if(n%2) j = -1;
for(k=0; k<=n; k++) {a[k] = j*p[k]; j *= -1;}
```

```

}

int sumDigits(unsigned long long n){
    int s = 0;
    while(n){
        s += n%10;
        n /= 10;
    }
    return s;
}

void write(unsigned long long a[31], short n){
    ofstream out("prodsum.out");
    short i;
    for(i=n-1; i>=0; i--){
        out << sumDigits(a[i]) << " ";
        out << a[i] << endl;
    }
}

int main(){
    unsigned long long a[31];
    short n;
    read(a, n);
    process(a, n);
    write(a, n);
    return 0;
}

```

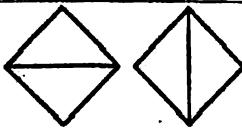
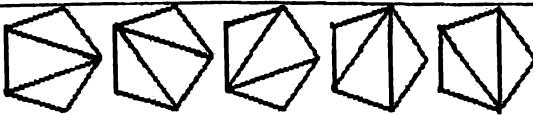
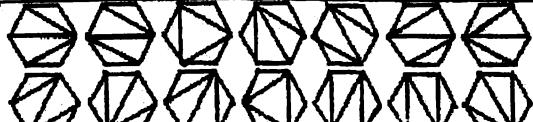
Probleme propuse

1. În program am folosit tablouri pentru a prezenta mai transparent algoritmul. Modificați-l astfel încât să folosiți în locul lor *vector*.
 2. Condiția ca numerele b_1, b_2, \dots, b_n să „încapă” în tipul *unsigned long long* nu mai este valabilă, ele pot fi oricât de mari. Dezvoltați programul, astfel încât să poată furniza rezultate și pentru aceste cazuri. Implementați clasa *NumarMare* care să opereze în mod natural cu astfel de numere și supraîncărcați operatorii aritmetici ($+, -, /, \%$, etc) de astfel că să fie întregită în mod natural în aritmética.
- Notă:* O implementare a numerelor mari în C se găsește, de exemplu, în [Log06], p. 377 (pe lângă operațiile uzuale conține și extragerea radicalului de ordin dat k dintr-un număr mare).

11 Programarea dinamică

Problema 12. Triangularizare minimală a unui poligon convex

Un poligon convex este un poligon ale cărui unghiuri sunt toate mai mici de 180° (în radiani, mai mici decât π). O triangularizare a unui poligon convex este împărțirea acestuia în triunghiuri cu ajutorul unor dintre diagonale, care nu se întrelap. Numărul de triangularizări al unui poligon convex cu $(n+2)$ vârfuri este al n -lea număr al lui Catalan: $C_n = \frac{1}{n+1} C_{2n}^n$:

	4 vârfuri → 2 posibilități
	5 vârfuri → 5 posibilități
	6 vârfuri → 14 posibilități

În general, pentru un poligon convex cu $n+2$ vârfuri, există C_n moduri să se triangularizeze poligonul astfel încât suma perimetrelor triunghiurilor constitutive să fie minimă (*triangularizare minimală*).

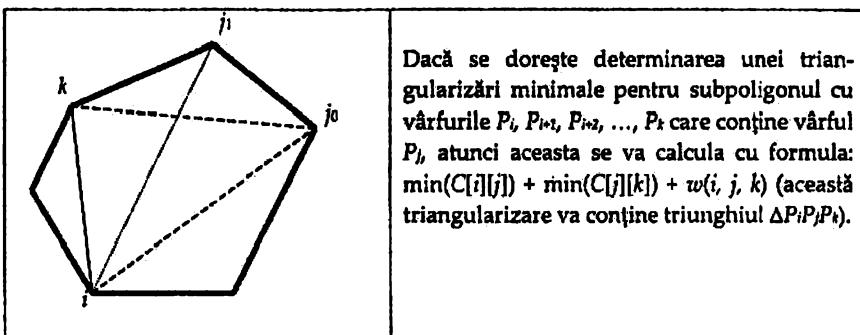
Date de intrare. În fișierul *triang.in* se găsesc mai multe puncte în plan, ce reprezintă abscisa și ordonata fiecărui vârf al poligonului, în ordinea inversă a celor de ceasornic.

Date de ieșire. În fișierul *triang.out* trebuie să fie date sumele perimetrelor triunghiurilor formate, apoi triangularizarea găsită. Exemplu:

<i>triang.in</i>	<i>triang.out</i>
-4 -9 8 8 -1 11 -8 8 -8 -5	Suma perimetrelor = 177,6 Triangularizarea: 0 1 5 1 4 5 1 2 4 2 3 4

Analiza problemei și proiectarea soluției

Notăm punctele date cu $P_0, P_1, P_2, \dots, P_{n-1}$. Salvăm perimetrele tuturor triunghiurilor posibile într-un vector w (formal $w(i, j, k)$ stochează perimetrul triunghiului cu vârfurile P_i, P_j și P_k). Suma perimetrelor triunghiurilor din care este formată o triangularizare reprezintă costul acesteia. Notăm cu $C[i][k]$, $0 \leq i < k \leq n-1$ costurile triangularizărilor minime pentru subpoligoanele cu vârfurile $P_i, P_{i+1}, P_{i+2}, \dots, P_k$. Răspunsul la problema noastră va fi deci $C[0][n-1]$.



Costurile triangularizărilor minime ale tuturor subpoligoanelor se vor calcula iterativ. Se începe cu subpoligoanele cu cel mai mic număr de muchii și se sfârșește cu subpoligonul cu număr n de vârfuri. De exemplu, dacă avem un poligon de 5 vârfuri, subpoligonul definit de muchii ce perechesc un vârfuri P_i și P_k , costul triangularizării minime este:

$$C[i][k] = \begin{cases} 0, & \text{pentru } k = i+1 \\ \min_{j=i+1}^k (\min(C[i][j]) + \min(C[j][k]) + w(i, j, k)), & \text{pentru } k > i+1 \end{cases} \quad (1)$$

Pe baza acestei formule, scriem metoda `doProcess()`, care calculează iterativ valorile $C[i][k]$ pentru toți i, k cu $0 \leq i < k \leq n-1$. Pentru salvarea costurilor minime, avem nevoie să adăugăm o matrice auxiliară $cost[i][k]$ unde $cost[i][k]$ va conține costul triangularizării minime a subpoligonului P_i, P_{i+1}, \dots, P_k . Vom folosi zona de dedesubtul acesta pentru a marca drumul spre triangularizarea corespunzătoare. Metoda care va lista triunghiurile constituente va folosi recursiv această zonă.

ALGORITHM_COST_TRIANGULATIONS

```

For ( $p = 1; p < n; \text{step } 1$ ) Execute // dimensiune subpoligon
    For (all pairs  $(i, k) \leftarrow (0, p), (1, p + 1), \dots, (n - p - 1, n - 1)$ ) Execute
        Calculate_with_Formel_1( $C[i][k]$ )
         $C[k][i] \leftarrow j$ , a.i.  $C[i][k] = \min_{i < j < k} \{C[i][j] + C[j][k] + w(i, j, k)\}$ 
    End_For
End_For
END_ALGORITHM_COST_TRIANGULATIONS

```

Metoda recursivă care furnizează triangularizarea minimală este:

```

Write_Triangulation( $i, j$ )
    If ( $j - i > 1$ ) Then
         $k \leftarrow C[j][i]$ 
        Write_Triangle( $i, k, j$ )
        Write_Triangulation( $i, k$ )
        Write_Triangulation( $k, j$ )
    End_If
End_Write_Triangulation( $i, j$ )

```

Dacă $P_1(x_1, y_1)$ și $P_2(x_2, y_2)$ sunt două puncte în plan, atunci distanța dintre ele este: $|P_1P_2| = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ (distanță euclidiană). Această formulă este implementată în metoda *dist()*.

Metoda *perimeter()* calculează perimetrul unui triunghi și are ca parametrii vârfurile acestuia. Cu ajutorul celor n puncte citite, care se vor salva în vectorul *vP*, metoda *calculatePerimeters()* calculează perimetrele tuturor triunghiurilor posibile în ordine lexicografică a tripletelor vârfurilor: $(0, 1, 2), (0, 1, 3), \dots, (n - 3, n - 2, n - 1)$. Metoda *getW()* furnizează perimetrul unui triunghi dat. Dacă elorim determinarea tripletelor cu $i < j < k$ ce sunt mai mici lexicografic decât (i_0, j_0, k_0) , adică se află înainte lui (i_0, j_0, k_0) în vectorul *w*. Numărul acestor triplete este suma tuturor tripletelor cu $i < i_0$, a celor pentru care $i = i_0$ și $j < j_0$ și a celor pentru care $(i = i_0, j = j_0 \text{ și } k < k_0)$:

$$|\{(i, j, k) \mid i < i_0\}| + |\{(i, j, k) \mid i_0 = i, j < j_0\}| + |\{(i, j, k) \mid i_0 = i, j_0 = j, k < k_0\}|$$

Pentru un $i < i_0$ fixat, există C_{n-i-1}^2 triplete (numărul de perechi (j, k) cu $0 \leq j < k < n$, adică toate perechile (j, k) cu $j < k$ din mulțimea $\{i_0 + 1, i_0 + 2, \dots, n - 1\}$). Pentru $i = i_0$ și un j_0 fixat există $n - j_0 - 1$ triplete (anume pentru $k = j_0 + 1, j_0 + 2, \dots, n - 1$). Pentru $i = i_0$, $j = j_0$ și $k < k_0$ există $k_0 - j_0 - 1$ triplete.

Program

```
#include <vector>
#include <iostream>
#include <cmath>

using namespace std;

typedef pair<double, double> Point;

void readVertices(ifstream &in, vector<Point*>& vP) {
    double x, y;
    vP.clear();
    while(in && !in.eof()){
        if(in >> x >> y)
            vP.push_back(new Point(x, y));
    }
}

inline double sqr(double x){
    return x*x;
}

double dist(const Point P1, const Point P2){
    double x1, x2, y1, y2;
    x1 = P1.first; y1 = P1.second;
    x2 = P2.first; y2 = P2.second;
    
$$|PP_2| = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

}

double perimeter(const Point P1, const Point P2,
                  const Point P3) {
    return
        dist(P1, P2) +
        dist(P2, P3) +
        dist(P3, P1);
}

void calculatePerimeters(vector<Point*>& vP,
                        vector<double> &w){
```

11 Programarea dinamică

43

```
int i, j, k, n=vP.size();
w.clear();
for(i=0; i<n-2; i++)
    for(j=i+1; j<n-1; j++)
        for(k=j+1; k<n; k++){
            w.push_back(perimeter(*vP[i], *vP[j], *vP[k]));
        }
}

double getW(vector<double> &w, int i0,
            int j0, int k0, int n){
    int p=0;
    int i, j;
    for(i=0; i<i0; i++){
        p +=(n-i-1)*(n-i-2)/2;
    }
    for(j=i0+1; j<j0; j++){
        p += n-j-1;
    }
    p+=k0-j0-1;
    return w[p];
}

void doProcess(vector<double> &w,
               int n, double C[][100]){
    for(i=0; i<n; i++)
        for(j=i+1; j<n; j++)
            val = 0;
            for(int p=1; p<n; p++){
                for(i=0; i<n-p; i++){
                    k = i+p;
                    if(i==p) C[i][k]=0;
                    else{
                        j=i+1;
                        C[i][k] = C[i][j]+C[j][k]+getW(w, i, j, k, n);
                        C[k][i]=j;
                        for(j=i+2; j<k; j++){
                            C[i][j] = C[i][j]+getW(w, i, j, k, n);
                            if(val<C[i][k]) {C[i][k]=val; C[k][i]=j;}
                        }
                    }
                }
            }
}

void writeTriang(ofstream &out, double C[][100], int i, int j){
    if((j-i)>1){
```

Altfel, se poate „sări” peste toate tri-

pletele mai mici lexicografic:

```
int p=0;
int i2, j2, k2;
for(i2=0; i2<10; i2++)
    for(j2=i2+1; j2<n-1; j2++)
        for(k2=j2+1; k2<n; k2++)
            p++;
    for(j2=i0+1; j2<j0; j2++)
        for(k2=j2+1; k2<n; k2++)
            p++;
    n+=10-10-1
    cout<< w[p];
```

```

int k=C[j][i];
out << " " << i << " " << k << " " << j << endl;
writeTriang(out, C, i, k);
writeTriang(out, C, k, j);
}
}

int main(){
vector<Point*> vP;
vector<double> w;
int n;
double C[100][100];
ifstream in("triang.in");
ofstream out("triang.out");
readVertices(in, vP);
n=vP.size();
calculatePerimeters(vP, w);
doProcess(w, n, C);
out << " Suma perimetrelor = ";
out.precision(4);
out << C[0][n-1] << endl;
out << " triangularizarea: " << endl;
writeTriang(out, C, 0, n-1);
return 0;
}

```

- | |
|---|
| <ol style="list-style-type: none"> 1. Citeste varfurile (vP). 2. Calculeaza perimetre (vP, w). 3. Construieste $C[i][j]$ pe baza w, n. 4. Scrie $C[0][n-1]$. 5. Scrie triangularizare recursiv. |
|---|

Probleme propuse

1. Repetati programul pentru problema modificata, care determinarea unei triangularizari, pentru care suma diagonalelor care formeaza triangularizarea sa fie minima.
2. Scrieti un program care prezinta o simulare grafica cat mai sugestiva a determinarii unei triangularizari. De exemplu, se poate avea un plan pe care mouse-ului se adaugă puncte în plan; dacă punctul curent formează cu celelalte puncte un poligon convex, atunci se va actualiza poligonul. Cu drag-and-drop aplicat unui vîrf, poligonul se modifică și triangularizarea minimă este înclădită. Se va actualiza și numărul de triunghiuri formate.
3. Demonstrați că formula explicită pentru $getW(i_0, j_0, k_0)$ este:

$$getW(i_0, j_0, k_0) = C_{n-1}^3 + C_{n-k_0-1}^3 + C_{n-j_0-1}^2 + C_{n-i_0-1}^1.$$

Problema 13. Înmulțirea unui sir de matrice

Pentru a înmulți două matrice A și B este necesar ca numărul de coloane ale lui A să coincidă cu numărul de linii ale lui B . Dacă matricea A are m linii și n coloane, iar B are n linii și p coloane, atunci produsul lor $A \cdot B$ are m linii și p coloane. Determinarea matricei produs se face cu $m \cdot n \cdot p$ înmulțiri elementare. A și B se pot scrie astfel ($m, n, p \in \mathbb{N} \setminus \{0\}$):

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}, B = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1p} \\ b_{21} & b_{22} & \dots & b_{2p} \\ \dots & \dots & \dots & \dots \\ b_{n1} & b_{n2} & \dots & b_{np} \end{pmatrix}$$

Atunci, matricea-produs $C = A \cdot B$ are forma:

$$C = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1p} \\ c_{21} & c_{22} & \dots & c_{2p} \\ \dots & \dots & \dots & \dots \\ c_{m1} & c_{m2} & \dots & c_{mp} \end{pmatrix}$$

$$\sum_{k=1}^n c_{ik} a_{kj} = \sum_{k=1}^n (c_{1k} a_{1j} + c_{2k} a_{2j} + \dots + c_{mk} a_{mj}) = c_{1j} a_{1j} + c_{2j} a_{2j} + \dots + c_{mj} a_{mj}, \quad i \in \{1, \dots, m\}, j \in \{1, \dots, p\}.$$

Pentru a determina C , este nevoie de $m \times n \times p$ înmulțiri elementare (n operații elementare pentru fiecare element c_{ij} cu $1 \leq i \leq m$, $1 \leq j \leq p$). Pentru a putea înmulți sirul de matrice $A_0 A_1 \dots A_n$, trebuie să fie îndeplinită condiția ca numărul de coloane ale

$$\text{număr_coloane}(A_i) = \text{număr_liniilor}(A_{i+1}) \text{ pentru orice } i \in \{0, \dots, n - 1\}.$$

Numeșteți adăugările diferențiale.

Catalan: $C_n = \frac{1}{n+1} C_{2n}^n$, egal cu numărul de parantetizări a $n+1$ termeni:

$a(b(c(d)))$, $a((b(c)d))$, $(ab)(cd)$, $(a(bc))d$ $((ab)c)d$	$\boxed{4}$ 4 termeni \rightarrow 5 posibilități
--	---

a (b (c (d e))), a (b ((c d) e)) a ((b c) (d e)), a (((b (c d)) e) a (((b c) d) e), (a b) (c (d e)) (a b) ((c d) e), (a (b c)) (d e) (a (b (c d))) e, (a ((b c) d)) e ((a b) c) (d e), ((a b) (c d)) e ((a (b c)) d) e, (((a b) c) d) e	5 termeni \rightarrow 14 posibilități
---	---

Primii termeni din sirul lui Catalan sunt: $C_0 = 1$, $C_1 = 1$, $C_2 = 2$, $C_3 = 5$, $C_4 = 14$, $C_5 = 42$, $C_6 = 132$, $C_7 = 429$, $C_8 = 1430$, $C_9 = 4862$, $C_{10} = 16796$, $C_{11} = 58786$.

Notăm cu $A(m, n)$ o matrice cu m linii și n coloane. Dacă vom considera matricile $A_0(40, 30)$, $A_1(30, 10)$ și $A_2(10, 2)$, atunci există două posibilități de a obține produsul lor $A_0A_1A_2$:

- $(A_0A_1)A_2$ – are nevoie în total de $40 \times 30 \times 10 + 40 \times 10 \times 2 = 12800$ produse elementare;
- $A_0(A_1A_2)$ – are nevoie în total de $40 \times 30 \times 2 + 30 \times 10 \times 2 = 3000$ produse elementare; aceasta înseamnă 23,44% din 12800. Dacă numărul de matrice crește, atunci diferențele dintre diferite posibilități cresc considerabil.

Pentru calculul produsului a opt matrice, există $C_7 = 429$ posibilități. Dacă avem, de exemplu, matricile $A_0(34, 23)$, $A_1(23, 12)$, $A_2(12, 2)$, $A_3(2, 5)$, $A_4(5, 80)$, $A_5(80, 3)$, $A_6(3, 3)$, $A_7(3, 12)$, atunci nu există $A_0A_1A_2A_3A_4A_5A_6A_7$ și produsul final este nedefinit.

- $(A_0(A_1((A_2A_3)A_4)))((A_5A_6)A_7)$ – are nevoie de 125800 produse elementare;
- $(A_0(A_1A_2))(((A_3(A_4A_5))A_6)A_7)$ – are nevoie de 4252 produse elementare, și aceasta înseamnă 3,38% din numărul primei parantetizări!

Problema ce apare este deci determinarea unei parantetizări pentru un sir de matrice, care furnizează un număr minim de înmulțiri elementare.

Date de intrare. În fișierul *matrix.in* se găsesc dimensiunile unui sir de matrice, și anume perechi (*număr linii, număr_coloane*), câte una pe fiecare linie. Se dau maxim 100 de matrice și numărul minim de produse elementare careape în tipul *unsigned long long*.

Date de ieșire. Scrieți pe prima linie a fișierului *matrix.out* numărul minim de produse elementare, iar pe următoarea linie o parantetizare corespunzătoare, ca în exemplu:

Date de intrare	Date de ieșire
34 23 23 12 12 2	minimal = 4252 (A0 (A1A2) (((A3 (A4A5)) A6) A7))

2 5
5 80
80 3
3 3
3 12

Analiza problemei și proiectarea soluției

Fie matricele A_0, A_1, \dots, A_n pentru care dimensiunile matricei A_i sunt d_i și d_{i+1} date. Introducem funcția $C(i, j)$, ce reprezintă numărul minim de produse elementare pentru calculul produsului șirului de matrice $A_i A_{i+1} \dots A_j$. Cu această notație, obținem:

- $C(i, i) = 0$, pentru orice $i = 0, \dots, n$ (1)
- $C(i, i + 1) = d_i \cdot d_{i+1} \cdot d_{i+2}$, pentru orice $i = 0, \dots, n - 1$ (2)
- $C(i, j) = \min_{1 \leq k < j} (C(i, k) + C(k + 1, j) + d_i \cdot d_{k+1} \cdot d_{j+1})$, $0 \leq i < j \leq n$ (3)

$C(i, k)$ este numărul minim de produse elementare pentru subsecvența $A_i A_{i+1} \dots A_k$. Rezultatul este o matrice cu d_i linii și d_{k+1} coloane. $C(k + 1, j)$ este numărul minim de produse elementare pentru subsecvența de matrice $A_{k+1} A_{k+2} \dots A_j$. Rezultatul este o matrice cu d_{k+1} linii și d_{j+1} coloane. Pentru a înmulți aceste două matrice rezultate, avem nevoie de încă $d_i \times d_{k+1} \cdot d_{j+1}$ produse elementare.

Vom calcula valorile iterativ pentru toate subșirurile de matrice de lungime 2, 3, ..., n . Deoarece valorile $C(i, j)$ se salvează în zona de deasupra diagonalei principale a tabloului bidimensional $C[i][j]$, zona de dedesubtul acestei nu este în dispozitie pentru a salva valoriile k ($C[j][i]$) și corespunzătoare k este condus la minimul $C[i][j]$). Pe baza formulelor (1), (2) și (3) scriem funcția `doProcess()`, care completează tabloul bidimensional $C[i][j]$. Dacă avem n matrice, atunci o parantezare minimală are $C[0][n-1]$ înmulțiri elementare. Pentru a reconstrui parantezarea minimală, scriem metoda `constructOrder()`, care completează tablourile `op[]` și `cl[]` cu semnificațile:

- $op[i]$ – numărul de paranteze deschise în fața matricei A_i ;
- $cl[i]$ – numărul de paranteze închise după matricea A_i .

Pentru o pereche dată (i, j) , valoarea corespunzătoare k din formula (3) este salvată în $C[j][i]$. În funcția `doProcess()` să se verifice că $i < j$ și că $C[i][j]$ sunt astfel parantezate:

```

int k=C[j][i];
if(i!=k){
    cout<<"(";
}
if(k+1!=j){
    op[k+1]++;
    cl[j]++;
}

```

și aceste secvențe vor fi optimal parantetizate:

```
| constructOrder(C, i, k, op, cl);
| constructOrder(C, k+1, j, op, cl);
```

În programul principal, se vor folosi tablourile `op[]` și `cl[]` pentru a reconstrui întreaga parantetizare.

Program

```
#include <fstream>
#include <vector>

using namespace std;

void read(ifstream &in, ofstream& out, vector<unsigned>& d) {
    int i, j;
    while(in && !in.eof()){
        if(in >> i >> j){
            if(d.size()>0 && d[d.size()-1]!=i){
                out << "Wrong input data: ";
                exit(1);
            }
            if(d.size()==0) (d.push_back(i), d.push_back(j));
            else {
                d.push_back(j);
            }
        }
    }
}

void doProcess(vector<unsigned> &d,
               unsigned long long C[][101]){

    int n = d.size()-1;
    for(i=1; i<n; i++){
        for(j=i+1; j<n; j++)
            C[i][j]=0;
        for(i=0; i<n-1; i++) C[i][i+1]=d[i]*d[i+1]*d[i+2];
        for(p=2; p<n; p++){
            for(i=0; i<n-p; i++){
                i+=1;
                C[i][j]=C[i][j]+C[i+1][j]+C[i+1][i+1]+C[i+1][i+2]+C[i+1][i+3];
                C[j][i]=i;
                for(k=i+1; k<j; k++){
                    aux = C[i][k]+C[k+1][j]+d[i]*d[k+1]*d[j+1];
                    if(C[i][j]>aux){C[i][j]=aux; C[j][i]=k;}
                }
            }
        }
    }
}
```

```
        }
    }
}

void constructOrder(unsigned long long C[1][101],
                     int i, int j, short op[100],
                     short cl[100]){
    if((j-i)>1){
        int k=C[j][i];
        if(i!=k){
            op[i]++; cl[k]++;
        }
        if(k+1!=j) {
            op[k+1]++; cl[j]++;
        }
        constructOrder(C, i, k, op, cl);
        constructOrder(C, k+1, j, op, cl);
    }
}

int main(){
    ifstream in("matrix.in");
    ofstream out("matrix.out");
    int n, i, j;
    vector<unsigned> d;
    for(i=0; i<n; i++)
        d.push_back(i);
    short op[100], cl[100];
    read(in, out, d);
    n=d.size()-1;
    for(i=0; i<n; i++){
        op[i]=0;
        cl[i]=0;
    }
    doProcess(d, C);
    out << "minimal = " << C[0][n-1] << endl;
    constructOrder(C, 0, n-1, op, cl);
    for(i=0; i<n;
        for(j=0; j<op[i]; j++) out<<"(";
        out << "A" << i;
        for(j=0; j<cl[i]; j++) out<<")";
    }
}
```

Probleme propuse

1. Dezvoltați programul astfel încât să se scrie în fișierul de ieșire și parantetizarea maximală, plus procentul cu 4 zecimale exacte al reducerii celei minimele în comparație cu cea maximală, ca în exemplu:

matrix.in	matrix.out
34 23 23 12 12 2 2 5 5 80 80 3 3 3 3 12	minimal = 4252 (A0(A1A2))(((A3(A4A5))A6)A7) maximal = 125800 (A0(A1((A2A3)A4)))((A5A6)A7) 3.38 %

2. Scrieți un program C++ care analizează suma timpilor de execuție pentru diferite date de intrare. Posibilitate de implementare: mai întâi, clasa *CMatrix*, ce reprezintă conceptul de matrice din matematică cu o serie de operatori (printre care citire, scriere, înmulțire, adunare etc.); se vor genera aleatoriu șiruri de matrice cu condiția din enunț și pentru fiecare se va calcula parantetizarea minimală și maximală. Aceste parantetizări, plus cea inițială se vor utiliza pentru a calcula produsul lor și, pentru fiecare, timpul de calcul se vor aduna unor

fiecare caz șirul de matrice, apoi parantetizările minimală și maximală, produsul lor obținut cu fiecare dintre ele, plus ordinea inițială (trebuie să fie același), timpul de execuție pentru determinarea fiecărui produs. La sfârșit, se vor afișa timpul de execuție globali, adică sumele pentru parantetizările minimală, maximală, inițială

Notă. Un program C care calculează produsul a două matrice se găsește, de exemplu, în [Log06], p. 145.

Problema 14. Edit-Distance

Metrica în matematică este o noțiune abstractă și nu se referă neapărat la distanța în spațiu (geometrică), care este o metrică particulară. O funcție $d: M \times M \rightarrow \mathbb{R}$

- $d(x, y) \geq 0$ pentru orice $x, y \in M$, $d(x, y) = 0 \Leftrightarrow x = y$;
- simetria: $d(x, y) = d(y, x)$ pentru orice $x, y \in M$;
- egalitatea triunghiului: $d(x, y) \leq d(x, z) + d(z, y)$ pentru orice $x, y, z \in M$.

O altă metrică cunoscută este *distanța Hamming*. Ea se definește ca fiind numărul de poziții în care două cuvinte de aceeași lungime nu coincid, atunci când se compară caracterele de pe aceste poziții. Exemplu: *distanța Hamming* pentru cuvintele *abcd* și *amcn* este 2 pentru că pe două poziții (2 și 4) caracterele nu coincid. Nu este greu să se demonstreze că *distanța Hamming* este o metrică.

O altă metrică este *Edit-Distance* (distanța de editare), numită și *Levenshtein-Distance* (după omul de știință rus Vladimir Levenshtein [n. 1935], care a prezentat algoritmul în 1965). Această distanță este definită pentru cuvinte formate pe un alfabet dat. Asupra unui cuvânt se pot folosi trei feluri de transformări: stergerea sau introducerea unui caracter, înlocuirea unui caracter cu un altul. *Edit-Distance* pentru două cuvinte W_1 și W_2 este definită ca fiind numărul minim de transformări necesare pentru a obține W_2 din W_1 . Exemplu: $d(\text{anne}, \text{marie}) = 3$, deoarece cuvântul *anne* se transformă în minim trei pași în *marie*: *anne* → *manne* → *marne* → *marie* (inserare: poziția 1; înlocuire: pozițiile 3 și 4).

Edit-Distance satisfac cele trei proprietăți ale unei metriki. Ea se folosește în practică printre altele la: prelucrarea și verificarea corectitudinii textelor, analiza ADN-ului, recunoașterea plagiilor, *File Revision, Remote Screen Update Problem*.

În această problemă, se cere determinarea distanței de editare dintre două cuvinte date. Cuvintele sunt formate din literele mici de la *a* la *z* și au lungimea maximă 100. Date de intrare. În fișierul *edit.in* se găsesc mai multe perechi de cuvinte, căte una pe linie.

Date de ieșire. Scrieți în fișierul *edit.out* distanța de editare pentru fiecare astfel de perechi de cuvinte. Exemplu: *edit.in* și *edit.out*

<i>edit.in</i>	<i>edit.out</i>
<i>anne marie</i>	$d(\text{anne}, \text{marie}) = 3$
<i>klipp klar</i>	$\text{anne I(1)} \rightarrow \text{manne T(3)} \rightarrow \text{marne T(4)} \rightarrow \text{marie}$
<i>iute dulce</i>	$d(\text{klipp}, \text{klar}) = 3$ $\text{klipp D(3)} \rightarrow \text{klipp T(3)} \rightarrow \text{klap T(4)} \rightarrow \text{klar}$ <hr/> $d(\text{iute}, \text{dulce}) = 3$ $\text{iute U(1)} \rightarrow \text{duce L(3)} \rightarrow \text{diuce L(4)} \rightarrow \text{dulce}$ <hr/> $d(\text{graba}, \text{treaba}) = 2$ $\text{graba T(1)} \rightarrow \text{traba I(3)} \rightarrow \text{treaba}$

Analiza problemei și proiectarea soluției

Fie cuvintele $x = x_1x_2\dots x_m$ și $y = y_1y_2\dots y_n$ de lungimi m , respectiv n . Dacă nu există litere comune în cele două cuvinte, atunci distanța de editare este $\max(m, n)$. Această distanță este mai mică atunci când există litere comune. O modalitate de rezolvare este construirea unui tablou bidimensional $T[i][j]$ cu $n+1$ linii și $m+1$ coloane cu semnificația că $T[i][j]$ este *Edit-Distance* dintre cuvintele $x_1x_2\dots x_i$ ($0 \leq i \leq m$) și $y_1y_2\dots y_j$ ($0 \leq j \leq n$). La început se inițializează linia 0 cu valorile $0, 1, 2, \dots, m$ (pentru a transforma cuvântul $x_1x_2\dots x_i$ în cuvântul vid, avem nevoie de j operații de stergere). Coloana 0 se va inițializa cu valorile $0, 1, 2, \dots, n$ (pentru a transforma cuvântul vid în cuvântul $y_1y_2\dots y_i$, avem nevoie de i operații de inserție). Pentru a obține distanța $T[i][j]$ între cuvintele $x_1x_2\dots x_i$ ($1 \leq i \leq j$) și $y_1y_2\dots y_i$ ($1 \leq i$), există trei posibilități:

- din $T[i-1][j-1]$: în cazul $x_i = y_i$, nu este nevoie de nici o transformare și putem avea $T[i][j] = T[i-1][j-1]$; dacă $x_i \neq y_i$, atunci x_i se înlocuiește cu y_i și putem avea $T[i][j] = T[i-1][j-1] + 1$;
- din $T[i-1][j]$: $y_1y_2\dots y_{i-1}$ este obținut din $x_1x_2\dots x_i$ cu $T[i-1][j]$ operații, adică $y_1y_2\dots y_i$ se obține din $x_1x_2\dots x_i$ prin $T[i-1][j] + 1$ operații, atunci când ultima dintre ele este inserarea literei y_i ;
- din $T[i][j-1]$: numărul minim de pași pentru a obține $y_1y_2\dots y_i$ din $x_1x_2\dots x_{i-1}$ este $T[i][j-1]$, adică $y_1y_2\dots y_i$ se obține din $x_1x_2\dots x_i$ prin $T[i][j-1] + 1$ operații, dacă ultima dintre ele este stergerea literei x_i .

Notăm $b(i, j) = 0$, dacă $x_i = y_i$, și $b(i, j) = 1$, dacă $x_i \neq y_i$. Atunci:

$$T[i][j] = \begin{cases} T[i-1][j-1] & \text{dacă } x_i = y_i \\ T[i-1][j] & \text{dacă } j = 0, i = 1, \dots, n \\ \min(T[i-1][j-1] + b(i, j), T[i-1][j] + 1, T[i][j-1] + 1), & \text{dacă } i = 1, \dots, n, j = 1, \dots, m \end{cases}$$

În continuare urmăreștem să calculăm distanța dintre cuvintele $x = kirk$ și $y = rump$. În figura următoare, două exemple, tabloul bidimensional $T[i][j]$ arată astfel:

	<i>a</i>	<i>n</i>	<i>n</i>	<i>e</i>
<i>a</i>	0	1	2	3
<i>r</i>	1	2	3	4
<i>i</i>	2	3	3	4
<i>k</i>	3	4	4	5

	<i>k</i>	<i>r</i>	<i>i</i>	<i>p</i>	<i>u</i>
<i>k</i>	0	1	2	3	4
<i>r</i>	1	2	3	4	5
<i>i</i>	2	3	3	4	5
<i>p</i>	3	2	1	2	3
<i>u</i>	4	3	2	1	2

Exemple Edit-Distance

Metoda `getEditDistance()` construiește tabloul $T[][]$. Valorile $T[i][j]$ vor fi iterativ calculate, linie cu linie, de la stânga spre dreapta, folosind formula de mai sus. Pentru a găsi apoi șirul de transformări, pornim cu $T[n][m]$ și ne deplasăm în sens invers până la câmpul $T[0][0]$. La fiecare pas $T[i][j]$ decidem care dintre cele trei operații a condus la obținerea valorii respective și stabilim una dintre ele, $T[i][j]$ a fost obținut pe baza uneia dintre celulele vecine: de pe diagonală deasupra-stânga, de deasupra sau din stânga. Vom folosi pentru fiecare dintre ele operația inversă pe cuvântul actual. O stergere devine inserție și invers, înlocuirea se face în sens invers.

Metoda recursivă `writeSolution()` realizează aceste operații. Ea se va apela cu cuvântul-scop ca parametru și determină o procedură de obținere a sa din cuvântul inițial pe baza tabloului $T[][]$. Metoda modifică cuvântul actual, se apelează recursiv cu noul parametru și scrie după aceea operația executată și cuvântul curent în fișierul de ieșire. Pot să existe mai multe drumuri optime, noi vom reconstrui însă doar unul dintre ele.

Dacă de exemplu $x_n = y_n$ și $T[n-1][m-1] = T[n][m]$, atunci $T[n][m]$ este obținut din $T[n-1][m-1]$ fără aplicarea vreunei operații, caz în care setăm variabila `flag` pe `true` (am găsit un predecesor pentru $T[n][m]$) și apelăm metoda `writeSolution()` cu noi parametri:

```
flag = true;
writeSolution(m-1, n-1, T, x, y, yaux, out);
```

Dacă $x_m \neq y_m$ și $T[n-1][m-1] + 1 = T[n][m]$, atunci $T[n][m]$ se obține din $T[n-1][m-1]$ prin adăugarea unei inserții (stergere și următoare). În acest caz setăm variabila `flag` pe `false` (nu am predecesor pentru $T[n][m]$). Aplicăm pe cuvântul curent y_m operația inversă, apelăm metoda `writeSolution()` cu noi parametri și scriem operația executată în fișierul de ieșire:

```
yaux.replace(n-1, 1, x.substr(m-1, 1));
if (!flag)
    writeSolution(m-1, n-1, T, x, y, yaux, out);
out << yaux << " T(" << n << ") --> ";
```

Același procedeu se folosește și pentru cazul când predecesorul lui $T[n][m]$ este $T[n-1][m]$ (adică $T[n-1][m-1]$ este predecesorul lui $T[n-1][m]$ căci $T[n][m] = T[n][m] - T[n-1][m] + 1$): transformare inversă, apel `writeSolution()` cu noi parametri, scrierea operației inverse. Remarcăm că în C++ literele șirului încep cu indexul 0; de aceea, testăm, de exemplu, $if(x[m-1]==y[n-1]...)$, în loc de $x[m]==y[n]$. În aceeași

Program

```

#include <string>
#include <fstream>

using namespace std;

short getEditDistance(string x, string y, short T[]){ //T[i][j] = distanta edit de la subsecventa x[0:i] si y[0:j]
    short m, n, i, j, res;
    m=x.length(); n=y.length();
    if(!m) res = n;
    if(!n) res = m;
    if(m && n){
        for(i=0; i<=m; i++) T[0][i]=i;
        for(i=0; i<=n; i++) T[i][0]=i;
        for(i=1; i<=n; i++)
            for(j=1; j<=m; j++){
                if(x[j-1]==y[i-1]){
                    T[i][j]=T[i-1][j-1];
                }
                else{
                    T[i][j]=T[i-1][j-1]+1;
                }
                if(T[i][j]>T[i-1][j]+1) T[i][j]=T[i-1][j]+1;
                if(T[i][j]>T[i][j-1]+1) T[i][j]=T[i][j-1]+1;
            }
        res = T[n][m];
    }
}

void writeSolution(int m, int n, short T[], string& x, string& y, string yaux, ofstream &out){
    if(m || n){
        if(m>0 && n>0){
            if(x[m-1]==y[n-1]){
                if(T[n-1][m-1]==T[n][m]){
                    writeSolution(m-1, n-1, T, x, y, yaux, out);
                }
            } else{
                if(T[n-1][m-1]+1==T[n][m]){
                    flag = true;
                    writeSolution(m-1, n-1, T, x, y, yaux, out);
                    out << yaux << " T(" << n << ") --> ";
                }
            }
        }
    }
}

```

```
        }
    );
}

if(n>0 && !flag){
    if(T[n][m]==T[n-1][m]+1){
        flag=true;
        yaux.erase(n-1, 1);
        writeSolution(m, n-1, T, x, y, yaux, out);
        out << yaux << " I(" << n << ") --> ";
    }
}

if(m>0 && !flag){
    if(T[n][m]==T[n][m-1]+1){
        yaux.insert(n, x.substr(m-1, 1));
        writeSolution(m-1, n, T, x, y, yaux, out);
        out << yaux << " D(" << n+1 << ") --> ";
    }
}
}

int main(){
    string x, y;
    cin >> x >> y;
    ifstream in("edit.in");
    ofstream out("edit.out");
    while(in && !in.eof()){
        if(in >> x >> y){
            res = getReditDistance(x, y);
            string yaux = y.substr(0, y.length());
            out << endl;
            out << "d(" << x << ", " << y << ") = "
                << res << endl;
            writeSolution(x.length(), y.length(), 0,
                          "", y, yaux, out);
            out << y << endl << "=====";
        }
    }
}
```

Probleme propuse

- Modificați metoda `writeSolution()` astfel încât să se afișeze toate modalitățile de a obține distanța de editare minimă.
- Presupunem că pentru cele trei operații sunt asociate costuri diferite: pentru înlocuire C_i , pentru stergere C_i , pentru inserarea unei litere C_i . Aceste costuri sunt date în fișierul de intrare ca triplete $(C_i, C_i \text{ și } C_i)$ imediat după perechile de cuvinte. Dezvoltați programul astfel încât să se afișeze o modalitate de transformare ce are cost minim. Exemplu:

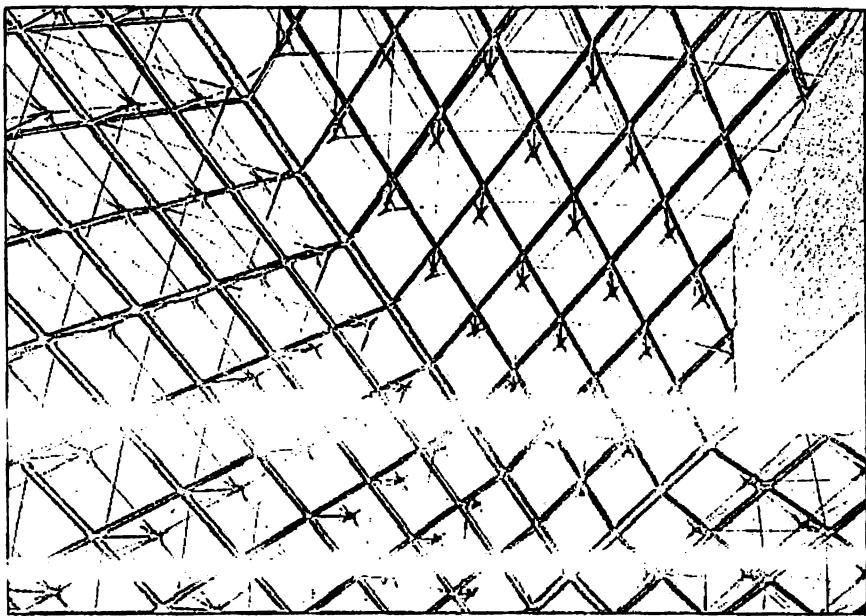
editi.in	editi.out
<pre> probleme loesungen 4 2 1 baerchen zwerglein 5 3 1 boese gute 2 3 1 </pre>	<pre> d(probleme, loesungen) = 17 probleme D(1) --> robleme D(1) --> obleme D(1) --> bleme D(1) --> leme I(2) --> loeme D(4) --> loes I(4) --> loese I(5) --> loesue I(6) --> loesune I(7) --> loesunge I(9) --> loesungen ===== d(baerchen, zwerglein) = 19 baerchen D(1) --> aerchen D(1) --> erchen I(1) --> zerchen I(2) --> zwerchen D(5) --> zwerhen D(5) --> zweren I(5) --> zwergen I(6) --> zwerglen I(8) --> zwerglein ===== d(boese, gute) = 7 boese D(1) --> oese T(1) --> gese T(2) --> guse T(3) --> gute =====</pre>

Modificați programul astfel încât să afișeze toate modalitățile de a obține acest cost minim, pentru cazul că există mai multe.

- Notăm cuvântul vid cu λ . Putem folosi și un algoritm de tip *Divide et Impera* bazat pe următoarele formule:

- $d(\lambda, \lambda) = 0,$
- $d(x_1 x_2 \dots x_i, \lambda) = i$ pentru orice $i \geq 1;$
- $d(\lambda, y y_2 \dots y_j) = j$ pentru orice $j \geq 1.$

$$d(x_1 x_2 \dots x_m, y_1 y_2 \dots y_n) = \begin{cases} \min_{x_i=y_j} (d(x_1 \dots x_{i-1}, y_1 \dots y_{j-1}) + d(x_{i+1} \dots x_m, y_{j+1} \dots y_n)), & \text{dacă există cel puțin o literă comună} \\ \infty, & \text{în caz contrar} \end{cases}$$



Piramida de stică de la Luvru, Paris

Bibliografie

- [Aig02] Aigner, Martin; Ziegler, Günter M., *Das Buch der Beweise*, Springer Verlag, Heidelberg, 2002.
- [Ban83] Banea, Horia, *Probleme de matematică traduse din revista sovietică Kvant*, Editura Didactică și Pedagogică, București, 1983.
- [Bar93] Barnsley, M.F., *Fractals everywhere*. Ediția a doua, Academic Press, Boston, 1993.
- [Bur94] Stan, M.; Burleson, W., „Limited-weight codes for low-power I/O”, *Int'l Workshop on Low Power Design*, 1994.
- [Coo71] Cook, S.A., „The complexity of theorem-proving procedures”, *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pp. 151-158.
- [Cor00] Cormen, Th.H.; Leiserson, C.E.; Rivest, R.; Stein, C., *Introducere în algoritmi*, Editura Computer Libris Agora, Cluj-Napoca, 2000.
- [Drè99] Drechsler, N.; Drechsler, R., „Exploiting don't cares during data sequencing using genetic algorithms”, *ASP Design Automation Conference*, pp. 303-306, 1999.
- [Gar79] Garey, M.R.; Johnson, D.S., *Computers and Intractability: a Guide to the Theory of NP-Completeness*, W.H. Freeman & Co., New York, 1979.
- [GI] *** Seria *Gazeta de Informatică* (1991-2006).
- [GM] *** Seria *Gazeta de Matematică* (1981-2000).
- [Gra94] Grattan-Guinness, Ivor, *Mathematics in Western Culture*, Oxford Mathematics, Addison-Wesley, 1994.
- [Gri98] *Povestile fraților Grimm*, ediție integrală, traducere de Viorica S. Constantinescu, Editura Polirom, Iași, 1998.
- [Iva98] Ivașc, Cornelia; Prună, Mona, Mateescu, E., *Programe informații (Concurs cu elemente de combinatorică) – Caiet de laborator*, Editura Petrion, București, 1998.
- [Iva99] Ivașc, Cornelia; Prună, Mona, *Tehnici de programare. Aplicații*, Editura Petrion, București, 1999.
- [Iva01] Ivașc, Cornelia; Prună, Mona, Mateescu, E., *Programare C++, Logofătu, Doină, Informatica C++. Manual pentru clasa a XI-a*, Editura Petrion, București, 2002.
- [Kar75] Karp, R.M.; „On the computational complexity of combinatorial problems”, *Networks*, 1975.
- [Log01] Logofătu, Doină, C++: *Probleme rezolvate și algoritmi*, Editura Polirom, Iași, 2001.

- [Log02] Logofătu, Doina, „Folosirea complementului și a reordonării pentru minimizarea tranzițiilor. Abordări evolutive”, referat prezentat la Universitatea „A.I. Cuza”, Iași, Facultatea de Informatică.
- [Log05i] Logofătu, Doina, *Suma puterilor asemenea*, *GInfo*, 15.02.2005, pp. 40-43, http://www.ginfo.ro/revista/15_2/focus2.pdf.
- [Log05ii] Logofătu, Doina, „Programare orientată obiect: de la o problemă de codificare la elemente POO cu C++”, *GInfo*, pp. 40-44, 15.04.2005, http://www.ginfo.ro/revista/15_4/focus3.pdf.
- [Log05iv] Logofătu, Doina, „De la problema cutiilor speciale la elemente POO cu C++”, *GInfo*, 15.05.2005, pp. 27-30, http://www.ginfo.ro/revista/15_5/focus1.pdf.
- [Log06] Logofătu, Doina, *Bazele programării în C. Aplicații*, Editura Polirom, Iași, 2006.
- [Log06i] Logofătu, Doina; Drechsler, Rolf, „Efficient Evolutionary Approaches for the Data Ordering Problem with Inversion”, *3rd European Workshop on Hardware Optimisation Techniques (EvoHOT)*, LNCS 3907, pp. 320-331, Budapest, 2006.
- [Log06ii] Logofătu, Doina, „Greedy Approaches for the Data Ordering Problem with and without Inversion”, *Proceedings of the ROSYCS 2006, Romanian Symposium on Computer Science*, pp. 65-80, Iași.
- [Log06iii] Logofătu, Doina, *Problema ordonării datelor cu și fără inversiune*, *GInfo*, 15.05.2006, pp. 8-14.
- [Log06iv] Logofătu, Doina, *Algorithmen und Problemlösungen mit C++*, Vieweg Verlag, Wiesbaden 2006.
- [Mun95] Murgai, R.; Fujita, M.; Krishnan, S.C., „Data sequencing for minimum transition”, *Int'l Workshop on Logic Synthesis*, 1995.
- [Mur97] Murgai, R.; Fujita, M.; Krishnan, S.C., „Data sequencing for minimum-transition transmission”, *IWIP Int'l Conference on VLSI*, 1997.
- [Mun98] Murgai, R.; Fujita, M.; Oliveira, A., „Using complementation and resequencing to minimize transitions”, *Design Automation Conference*, pp. 694-697, 1998.
- [Năs83] Năstăsescu, C.; Niță, C.; Brandiburo, M.; Ioniță, D., *Exerciții și probleme de matematică și fizică*, Editura Didactică și Pedagogică, București, 1983.
- [Năs96] Năstăsescu, C.; Niță, C.; Popa, S., *Algebra. Manual pentru clasa a X-a*, Editura Didactică și Pedagogică, București, 1996.
- [Năs97] Năstăsescu, C.; Niță, C.; Rizescu, Gh., *Algebra. Manual pentru clasa a IX-a*, Editura Didactică și Pedagogică, București, 1997.
- [Nit04] Nitzsche, Manfred, *Graphen für Einsteiger. Rund um das Haus von Nikolaus*, Vieweg Verlag, Wiesbaden, 2004.
- [Olt00] Oltean, Mihai, *Proiectarea și implementarea algoritmilor*, Computer Libris Agora, Cluj-Napoca, 2000.

- [Ran97] Rancea, Doina, *Limbajul Pascal. Manual clasa a IX-a*, Computer Libris Agora, Cluj-Napoca, 1997.
- [Ran99] Rancea, Doina, *Limbajul Pascal. Algoritmi fundamentali*, Computer Libris Agora, Cluj-Napoca, 1999.
- [Rec99] Rector, Brent; Sells, Chris, *ATL Internals*, Addison Wesley Longman, Inc., Massachusetts, 1999.
- [Sip97] Sipser, M., *Introduction to the Theory of Computation*, PWS-Kent, Belmont, California, 1997.
- [Sta06] „Standard Template Library Programmer’s Guide”, <http://www.sgi.com/tech/stl>.
- [Str04] Stroustrup, Bjarne, *Die C++ Programmiersprache*. Ediția a patra, revăzută, Auflage, Addison-Wesley, München, 2004.

Referințe online

- <http://de.wikipedia.org/wiki/Hauptseite>
- <http://acm.uva.es/problemset/>
- <http://ceoi.inf.elte.hu/>
- <http://linuxgazette.net/issue34/field.html>
- <http://mathworld.wolfram.com/>
- <http://microscopy.fsu.edu/optics/timeline/people/>
- <http://olympiads.win.tue.nl/iowl>
- <http://www.acm.org/>
- <http://www.maths.org/>
- <http://www.math-th-lind.org>
- <http://www.math.utah.edu/mathcircle/notes/miladen2.pdf>
- <http://www.mathematica.de>
- <http://www.mathematik.uni-ulm.de/obst/>
- <http://www-i1.informatik.rwth-aachen.de/~algorithmus>

Glosar

A

Ackermann, funcția 171
Active Template Library (ATL) 182
algoritm 14, 58
Algoritm Greedy Min (GM) 138
Algoritm Greedy Min Simplified (CMS) 138
Algoritm LB 138
ALGORITM_Backtracking_Iterativ 232
ALGORITM_Backtracking_Recursiv 233
ALGORITM_BFS 66
ALGORITM_CĂUTARE_BINARĂ 16
ALGORITM_COST_TRIANGULATIONS 335
ALGORITM_CUTII_ASCUNSE 80
ALGORITM_DFS 64
ALGORITM_DivideEtImpera 205
ALGORITM_EUCLID 14
ALGORITM_EXACT_POD 137
ALGORITM_EXACT_PODI 137
ALGORITM_Greedy 105
ALGORITM_GREEDY_COMPACT 270
ALGORITM_Huffman 123
ALGORITM_KRUSKAL 117
ALGORITM_LCS 297
ALGORITM_N_REGINE 230
ALGORITM_OPTIM_COMPACT_REC 267
ALGORITM_RAN 136
ALGORITM_RAN_POD 136
ALGORITM_SITA_ERATOSTENE 15
ALGORITM_SUME_PUTERI 96
Algorithmica 21
analiza complexității 26
algoritmul lui Kruskal (1956) 116, 139
analogie 10
arboare binar complet 120, 125
arbore parțial 115
arboare parțial binar 120
Arbore – toamnă în Ottawa 224
Axiomele lui Peano 157

B

Backtracking cu lungime variabilă 232, 234, 237
Backtracking în plan 232, 244, 246, 248
Backtracking iterativ 232, 234, 237

Backtracking liniar cu lungime fixă 232, 234, 237, 241, 260
Backtracking recursiv 233, 244, 246, 248, 252, 260, 263, 277
baze ascunse 276
Bellman, Richard (1920-1984) 285
Bernoulli, inegalitatea 162
Bezzel, Max (1824-1871) 227
BFS (Breadth First Search) 65, 322
Big Mod 167
Bin Packing Problem 30
binomul lui Newton 92, 159
Borland C++ 3.1, 181, 190

C

cal pe tablă de șah 112, 244, 320
Case oglindite în Lübeck 126
Căsuța lui Moș Crăciun 263
căutarea binară 16
Catalan, șirul lui 125, 320, 333, 339
Cauchy-Bunakowski-Schwarz, inegalitatea 160
Cel mai lung subșir comun (LCS) 296
Ciușul lui Eratostene 15, 71
clasi 40, 81, 107
clasificarea problemelor 28
cod binar 119
cod binar de lungime fixă 120
cod binar de lungime variabilă 120
codificarea Huffman 119
codificarea informației 19, 20
codificarea informației binare 90, 202
Collatz, funcția 176
colorarea hărții 19, 110, 242
compactarea şablonelor de test 265
complexitatea algoritmului 23, 24, 57, 116, 211, 212, 269
container 51
container adaptiv 52
container asociativ 51
container secvențial 51
container specializat 52
crearea obiectelor (constructori) 43, 81, 123, 222
curba lui Koch 194
cutie n-dimensională 77

D

- demonstrație directă 158, 160, 179
 Detaliu al unei săfârșiri publice din Zürich 33
DFS (Depth First Search) 64
DFT – transformata Fourier discretă 220
 distanță de editare 345
 distanță euclidiană 335
distanță Hamming 129, 345
 distrugerea obiectelor (destructori) 44, 222
Divide et Impera 205
 divizibilitate prin număr prim 159
 Domino 303

E

- Eratostene 15
 Euclid 14
 Euler, Leonhard 257

F

- factorial 68
 Fibonacci (Leonardo Pisano) 289
 șirul lui ~ 162, 286, 289
 Fractal space-filling 200
friend 82, 84, 222
 funcție binară 61
 funcție de cost a grafului 115
 funcție surjectivă 241
 funcție proprietate 61
 funcție urmărită 61

G

- garbage collection* 44
 generarea cuvintelor 143
 generarea aranjamentelor 226
 generarea combinațiilor 207
 generarea funcțiilor surjective 242
 generarea partițiilor 238
 generarea permutărilor 70
 generator 61

 Graffiti în München 281
Graph Colouring Problem 30
Greedy 105

- Hoare, C. A. R. (n. 1934) 211
 Huffman, David Albert (1925-1999) 120

I

- inducția matematică 157
inline 107, 109
 instanță a unei clase 40
 instrucțiunea de decizie *if*: 305
 Integrală cu regula trapezului 209
 inversarea unui cuvânt 129
 iterator 55
 intereschimbarea a două elemente 49
 Iulius Cezar 205

J

- împărțirea cadourilor 307
 încapsulare 41
 înmulțirea unui șir de matrice 339

K

- Knapsack Problem* (problema rucsacului) 300
 Koch, Helge von 194
 Kruskal, Joseph Bernard 116

L

- labyrinthul 277
 Lebede pe râul Isar în München 202
 Levenshtein, Vladimir 345
 lexicografic 70, 79, 263
Liber Abaci (Cartea Abacului) 289
 LIFO 110, 111
 Linie, șirul lui 291
 Ludovic al XI-lea 205

M

- Manna-Pnueli, funcția 172
Max-Cutting Problem 10, 176
 matricea de adiacență 110, 131
memoization 286
memory leak 44
MergeSort (sortarea prin interclasare) 213
 metričkă 130

- München – fan bavarez al Campionatului Mondial de Fotbal (Germania, 2006) 102

N

- lower bound* 23
 Notația – O (mărginire superioară – *upper bound*) 24
 Notația – Θ (categorie constantă – *same order*) 23

număr prim 61
numărul total de tranziții 131
numere complexe 221

O

operator- 222
operator() 109
*operator** 222
operator+ 222
operator< 107, 109, 124
operator<< 107, 222
operator> 82
operator>> 82, 107, 222
optimalitatea algoritmului 24

P

Partition Problem 30
Pătrate magice 227, 233
Peano, Giuseppe 157
permutare 61, 140
pioni pe tabla de săh 275
Piramida de sticlă de la Luvru 351
polimorfism 43
polinom, coeficienți 330
POO (engl. OOP) 37
Positano, Italia 46
predicție 61

R

principiu opozitiv 30
Problema celor patru rege 27
Problema colorării 234, 235, 236, 278
Problema comisului voiajor 28

S

Problema fotografiei 246
Problema ordonării datelor 127
Problema rucsacului 106
probleme NP-complete 29, 134
produs cartezian 2/4
produs de matrice 339
programare dinamică 77, 97, 285
programarea generică 49

Proprietații, proprietăți 221
proprietățile algoritmului 21
proprietățile proiectului 189

Q

Quad-Trees 215
QuickSort (sortarea rapidă) 211

R

rădăcinile unității 220
Random Access Machine (RAM) 22
recursivitate 163
recursivitate imbricată 163, 171
recursivitate deschisă 164, 176
recursivitate directă 164, 181
recursivitate indirectă 164, 190
recursivitate liniară 163, 169
recursivitate ramificată 163, 175
reducere la absurd 79
reducerea algoritmului 25
relațiile lui Viète 175, 330
ridicarea rapidă la putere 1:5

S

SAT – problema satisfiabilității 30
Schedule Problem 20
Scotienii la Oktoberfest 314
Set Cover Problem 30
Standard Template Library (STL) 49
static, element 44, 107
Statuile în Bremen 154

std::list 63, 107, 111, 120, 121, 123, 149, 231
std::copy 63
std::deque 51
std::find 65, 67, 147
std::find_if 62, 261
234, 235, 246
std::iostream 63, 206, 231
std::iterator 61, 63, 85
std::reverse_iterator 57
std::set 51, 143
std::stack 51, 143
std::map 51, 69
std::multimap 52, 117, 143
std::multiset 52, 69
std::multimap 52, 69
std::multiset 52, 69
std::ostream 63
std::pair 98, 118, 124, 143, 260, 304, 323
std::priority_queue 52, 109
std::queue 52, 66, 108

std::random_shuffle 58, 63, 144
std::set 52, 254
std::set 68, 260
std::sort 59, 82, 84, 99
std::stack 52, 64
std::string 52, 268
std::valarray 52
std::vector 51, 63, 64, 66, 70, 81, 143, 231, 239, 254,
 268, ...
std::vector<bool> 143
 subșir crescător maximal 79, 293
 succesor în șir de biți (*next_bitstring*) 141
 succesor 112
 Sudoku 257
 Sume asemănătoare 309
 sume de produse 329
 sume de puteri 39
 supraincărcarea operatorilor 82, 107, 109, 124,
 222

T
template 50, 182, 214
 transformare recursivă în altă bază 173
 triangularizarea minimală a unui
 poligon convex 333
 triunghi de numere 300
 triunghiul lui Pascal 292
 Triunghiul lui Sierpinski 199
 trucuri de programare 85
 Turnurile din Hanoi 207

T
 Tăranul, lupul, capra și varza 274-275

V
 Vedere din turnul catedralei din Ulm 73

W
 Warnsdorff, regula 112



Turnul din Pisa

www.polirrom.ro

Bun de tipar: februarie 2007. Apărut: 2007
Editura Polirrom, B-dul Carol I nr. 4 • P.O. Box 266, 700506, Iași
Tel. & Fax (0232) 21.41.00; (0232) 21.41.11;
(0232) 21.74.40 (difuzare); E-mail: office@polirrom.ro
P-4111 C, D.Mihailovici nr. 6, et. 7 nr. 33; O.P. 37 •

Tel.: (021)313.89.78, E-mail: polirrom@jdu.ro

Tiparul executat la S.C. LUMINA TIPO s.r.l.
str. Luigi Galvani nr. 20 bis, sect. 2, București

Tel./Fax: 211.32.60, 212.29.27, E-mail: lumina-tipo@fx.ro