

EMANUELA CERCHEZ, MARINEL ȘERBAN

**Programarea în limbajul C/C++
pentru liceu**



EMANUELA CERCHEZ (n. 1968, Iași) este absolventă a Facultății de Matematică, secția Informatică (1990), și a Seminarului pedagogic postuniversitar (1997), profesoră de informatică (grad didactic I), membră în Comisia Națională de Informatică. Autoarea a mai publicat la Editura Polirom : *Internet. Manual pentru liceu* (2000, avizat MEN), *Informatica. Manual pentru clasa a X-a* (coautor Marinel-Paul Șerban, 2000, avizat MEN), *Informatica. Manual pentru clasa a X-a* (coautor Marinel-Paul Șerban, 2001 ; ed. a II-a, 2005), *Informatica pentru gimnaziu* (coautor Marinel-Paul Șerban, 2002, avizat MEC), *Informatica. Culegere de probleme pentru liceu* (2002), (coautor Marinel-Paul Șerban, 2005), *Programarea în limbajul C/C++ pentru liceu. Metode și tehnici de programare* (vol. I, coautor Marinel-Paul Șerban, 2005), *Programarea în limbajul C/C++ pentru liceu. Metode și tehnici de programare* (vol. II, coautor Marinel-Paul Șerban, 2005), *Programarea în limbajul C/C++ pentru liceu* (vol. III, coautor Marinel-Paul Șerban, 2006).

MARINEL-PAUL ȘERBAN (n. 1950, Arad) este absolvent al Facultății de Matematică-Mecanică, Universitatea din Timișoara (1973); specializare postuniversitară în informatică (1974), profesor de informatică (grad didactic I), membru în Comisia Națională de Informatică. De același autor, la Editura Polirom au apărut : *Informatica. Manual pentru clasa a X-a* (coautoare Emanuela Cerchez, 2000, avizat MEN), *PC. Pas cu pas* (coautoare Emanuela Cerchez, 2001 ; ed. a II-a, 2005), *Informatica pentru gimnaziu* (coautoare Emanuela Cerchez, 2002, avizat MEC), *Programarea în limbajul C/C++ pentru liceu* (vol. I, coautoare Emanuela Cerchez, 2005), *Programarea în limbajul C/C++ pentru liceu. Metode și tehnici de programare* (vol. II, Cerchez, 2005), *Programarea în limbajul C/C++ pentru liceu. Metode și tehnici de programare* (vol. III, coautoare Emanuela Cerchez, 2006).

Autorii au o bogată experiență în pregătirea de performanță a elevilor : susțin cursuri la Centrul de Pregătire a Tinerilor Capabili de Performanță din Iași, propun probleme pentru olimpiadele și concursurile naționale și județene de informatică, susțin activitatea de pregătire a lotului național de informatică, precum și programul de pregătire de performanță în informatică .campion.

© 2013 by Editura POLIROM

Această carte este protejată prin copyright. Reproducerea integrală sau parțială, multiplicarea prin orice mijloace și sub orice formă, cum ar fi xeroxarea, scanarea, transpunerea în format electronic sau audio, punerea la dispoziția publică, inclusiv prin internet sau prin rețele de calculatoare, stocarea permanentă sau temporară pe dispozitive sau sisteme cu posibilitatea recuperării informațiilor, cu scop comercial sau gratuit, precum și alte fapte similare săvârșite fără permisiunea scrisă a deținătorului copyrightului reprezentă o încălcare a legislației cu privire la protecția proprietății intelectuale și se pedepsesc penal și/sau civil în conformitate cu legile în vigoare.

www.polirom.ro

Editura POLIROM
Iași, B-dul Carol I nr. 4, P.O. BOX 266, 700506
București, Splaiul Unirii nr. 6, bl. B3A, sc. 1, et. 1,
sector 4, 040031, O.P. 53, C.P. 15-728

Descrierea CIP a Bibliotecii Naționale a României :
CERCHEZ, EMANUELA

Programarea în limbajul C/C++ pentru liceu / Emanuela Cerchez, Marinel Șerban. – Iași : Polirom, 2005-2013

4 vol.

ISBN 978-973-46-0109-7

Vol. 4. *Programare orientată pe obiecte și programare generică cu STL. – 2013. – Bibliogr.*
ISBN 978-973-46-4081-2

I. Șerban, Marinel

004.42(075.35)

004.43 C(075.35)

004.43 C ++(075.35)

Printed in ROMANIA

matică (1990),
l), membră în
al pentru liceu
, 2000, avizat
ntru gimnaziu
liceu (2002),
rogramare în
şerban, 2005),

, Universitatea
rmatică (grad
m au apărut:
C. Pas cu pas
are Emanuela
are Emanuela
nare (vol. II,
III, coautoare

de Pregătire a
e naționale și
programul de

ea prin orice
ic sau audio,
ermanentă sau
al sau gratuit,
reprezintă o
i/sau civil în

Emanuela Cerchez, Marinel Şerban

PROGRAMAREA ÎN LIMBAJUL C/C++ PENTRU LICEU

• • •

Programare orientată pe obiecte
și programare generică cu STL

ăști :

ogr.

POLIRO
2013

Cu

1.

2.

CUPRINS

<i>Cuvânt înainte</i>	7
1. Principiile programării orientate pe obiecte.....	9
1.1. Evoluția limbajelor de programare.....	9
1.2. Principiile <i>POO</i>	12
1.3. Avantajele <i>POO</i>	13
1.4. Întrebări recapitulative	14
2. Programarea orientată pe obiecte în <i>C++</i>	15
2.1. Clasele și obiectele	15
2.2. Controlul accesului la membrii unei clase	16
2.3. Arhitectura unei aplicații <i>POO</i>	18
2.4. Definirea funcțiilor membre în exteriorul clasei	18
2.5. Funcțiile <i>inline</i>	19
2.6. Funcțiile cu parametri implicați	20
2.7. Suprăîncărcarea funcțiilor	21
2.8. Constructorii.....	22
2.9. Destructorul.....	25
2.10. Modelul logic al vieții unui obiect	26
2.11. Constructorul de copiere.....	28
2.12. Pointerul <i>this</i>	29
2.13. Membrii statici ai unei clase	29
2.14. Specificatorul <i>const</i>	32
2.15. Funcțiile <i>friend</i>	34
2.16. Clasele <i>friend</i>	35
2.17. Suprăîncărcarea operatorilor	36
2.18. Tratarea erorilor	48
2.19. Aplicație. Numere naturale mari	51
2.20. Clasa <i>string</i>	64
2.21. Moștenirea	70
2.22. Includerea condiționată	76
2.23. Polimorfismul	77
2.24. Tratarea erorilor utilizând clasa <i>exception</i>	81
2.25. Exerciții și probleme propuse.....	85

3.	Elemente de programare generică	96
3.1.	Functiile şablon	96
3.2.	Clasele şablon	99
3.3.	Exerciţii şi probleme propuse.....	104
4.	<i>STL</i> . Concepte generale	107
4.1.	Ce este <i>STL</i> ?	107
4.2.	Clasele container	108
4.3.	Clasele adaptor pentru containere	109
4.4.	Iteratorii	110
4.5.	Clasele adaptor pentru iteratori.....	112
4.6.	Functorii.....	114
4.7.	Clasele adaptor pentru functori	115
4.8.	Algoritmii	115
4.9.	Clasa şablon pair	121
4.10.	Alocatorii.....	123
4.11.	Aplicaţii	123
4.12.	Exerciţii şi probleme recapitulative	131
5.	Containere secvenţiale	133
5.1.	Clasa vector	133
5.2.	Clasa deque	147
5.3.	Clasa list	152
5.4.	Clasa forward_list	159
5.5.	Clasa array	162
5.6.	Exerciţii şi probleme propuse	163
6.	Clasele adaptor	170
6.1.	Clasa adaptor queue (coada).....	170
6.2.	Clasa adaptor stack (stiva).....	181
6.3.	Clasa adaptor priority_queue	188
6.4.	Exerciţii şi probleme propuse	200
7.	Containere asociative	207
7.1.	Containerele asociative sortate set şi multiset	207
7.2.	Containerele asociative sortate map şi multimap	220
7.3.	Containerele asociative nesortate	231
7.4.	Exerciţii şi probleme propuse	234
8.	Soluţii şi indicaţii	239
	<i>Anexă. Caracteristici ale principalelor containere</i>	246
	<i>Bibliografie</i>	247

.....	96
.....	96
.....	99
.....	104
.....	107
.....	107
.....	108
.....	109
.....	110
.....	112
.....	114
.....	115
.....	115
.....	121
.....	123
.....	123
.....	131
.....	133
.....	133
.....	147
.....	152
.....	159
.....	162
.....	163
.....	170
.....	170
.....	181
.....	188
.....	200
.....	207
.....	207
.....	220
.....	231
.....	234
.....	239
.....	246
.....	247

Cuvânt înainte

Volumul al IV-lea al lucrării *Programarea în limbajul C/C++ pentru liceu* reprezintă o continuare firească a celor anterioare, cel de față cuprinzând prezentarea a două noi paradigmă de programare: programarea orientată pe obiecte și programarea generică. Prezentarea concisă, clară și operațională a conceptelor programării orientate pe obiecte a reprezentat o adevărată provocare. Suntem convinși însă că am reușit să punem la dispoziție un instrument eficient de învățare, datorită numeroaselor exemple explicate și aplicațiilor de factură algoritmică incluse, utile atât elevilor de liceu, cât și studenților care se inițiază în programare.

Cea de-a doua secțiune a lucrării este axată pe programarea generică și ilustrarea acestui nou stil de programare, utilizând biblioteca *STL* (*Standard Template Library*). Nu ne-am propus să înlocuim în acest volum documentația *STL*, care este excelent structurată și poate fi accesată online, ci am urmărit ilustrarea principalelor funcții printr-o abordare algoritmică. Am integrat cunoștințe și deprinderi pe care elevii și le-au format deja prin studiul volumelor precedente, dar am schimbat perspectiva în ceea ce privește modul de implementare, valorificând facilitățile oferite de *STL*.

Prefața pe care Alexander Stepanov, principalul autor al bibliotecii *STL*, a scris-o în 2001 pentru lucrarea *STL Tutorial and Reference Guide* se încheie astfel: „*STL* presupune un mod diferit de a predă informatică. 99% din ceea ce un programator trebuie să știe nu este cum să construiească o componentă *software*, ci cum să o folosească”. Ca profesori cu zeci de ani de experiență în formarea programatorilor de performanță, am dori să nuanțăm această afirmație, deoarece suntem convinși că pentru a utiliza eficient și productiv o componentă *software* trebuie să ai o bună înțelegere a modului de funcționare a acesteia. Din acest motiv nu am putut prezenta biblioteca *STL* fără a ilustra principiile programării orientate pe obiecte și ale programării generice, precum și modul de implementare a acestor două stiluri de programare în limbajul *C++*. Și suntem convinși în egală măsură că „învățăceii” noștri, care stăpânesc deja structurile de date, vor înțelege perfect containerele standard din *STL* și vor să le utilizeze optim în implementările lor.

Sperăm ca prin acest volum să oferim un punct de sprijin în construirea unui răspuns afirmativ la întrebarea pe atunci retorică a lui Alexander Stepanov: „Putem muta vreodată *software-ul* în era industrială? ”.

a]
s.
a

C
B
a
A
c
E
p
h
A

e
l
a
c
c
s
i
c
c
1
c
1

1. Principiile programării orientate pe obiecte

1.1. Evoluția limbajelor de programare

Programarea orientată pe obiecte (pe scurt, *POO*) reprezintă un stil de dezvoltare aplicații (sau, mai elegant spus, o paradigmă de programare). Ca să înțelegem cum s-a ajuns la *POO*, ar fi utilă o scurtă incursiune istorică în evoluția calculatoarelor și a limbajelor de programare.

Prima mașină de calcul programabilă a fost concepută de matematicianul englez Charles Babbage, în perioada 1830-1840. Interesant este că modelul propus de Babbage se apropie de structura conceptuală a unui calculator actual (avea o unitate aritmetico-logică, o unitate de control, o unitate de memorie). Ada Lovelace (mai exact, Augusta Ada King, Contesă de Lovelace, născută Augusta Ada Byron, fiind singurul copil legitim al poetului George Gordon Byron) a fost primul programator din lume. Ea a conceput un „limbaj de programare” pentru mașina lui Babbage și a creat primul program din lume. Din păcate, unele probleme tehnice și financiare nu i-au permis lui Babbage să finalizeze construcția mașinii sale de calcul, astfel încât programul Adei Lovelace nu a fost niciodată testat.

O sută de ani mai târziu, între anii 1940 și 1950, au fost create primele calculatoare electronice. Cele mai importante teorii care au stat la baza construcției calculatoarelor le aparțin lui Alan Turing (care în 1936 a descris un model matematic de funcționare a unei mașini de calcul programabile), Claude Shannon (care în 1937 a demonstrat că orice funcție din algebra booleană poate fi implementată mecanic cu ajutorul unor circuite logice electronice) și John von Neumann (care a descris schema structurală de bază a calculatoarelor, denumită arhitectura von Neumann); calculatoarele moderne sunt construite din circuite logice, se bazează pe arhitectura von Neumann și implementează funcțional modelul mașinii Turing. La titlul de „primul calculator modern” candidață mașina Atanasoff-Berry (primul calculator electronic), calculatorul britanic Colossus (primul calculator electronic programabil prin conexiunile cablurilor și comutatoare), mașinile inginerului german Konrad Zuse (prima mașină Turing completă) și ENIAC (primul calculator generic; programul trebuia introdus manual, modificând conexiunile cablurilor și comutatoarele sale). Primul calculator funcțional cu program stocat a fost Manchester „Baby” dezvoltat de Frederic C. Williams și Tom Kilburn la University of Manchester în 1948. Odată cu dezvoltarea calculatoarelor cu programe stocate s-au dezvoltat și limbajele de programare.

Înțial, calculatoarele erau programate în limbaj mașină (programatorii cunoșteau codurile numerice ale instrucțiunilor specifice mașinii de calcul utilizate și le introduceau „de mâna”, proces, evident, lent și supus frecvent erorilor). O idee majoră, care a schimbat acest stil de programare, a fost aceea de a crea un program care să traducă un alt program. Astfel au apărut limbajele de asamblare. Acestea sunt de asemenea specifice mașinii pe care rulează, asamblorul traducând instrucțiunile limbajului de asamblare în cod mașină. Instrucțiunile scrise în limbaj de asamblare au o codificare mai ușor de reținut (*mnemonic*) pentru operația executată și operanzi (date sau adrese de memorie). De exemplu, MOV AL 61h determină memorarea valorii hexazecimale 61 în registrul de memorie AL.

Ideeua care a făcut trecerea de la limbajele de asamblare (dificil de utilizat și specifice mașinii de calcul) la limbajele de nivel înalt a fost introdusă în 1952 de Grace Hopper : utilizarea unui limbaj de programare independent de mașină și a unui program (pe care ea l-a denumit compilator) prin care să fie tradus codul-sursă (codul scris în limbajul de programare respectiv) în cod obiect (cod scris în limbaj mașină). De altfel, Grace Hopper a și dezvoltat primul compilator pentru limbajul A-0, deși acest merit este atribuit frecvent echipei conduse de John Backus de la IBM, care a dezvoltat în 1957 compilatorul pentru limbajul FORTRAN. Din acest moment s-au dezvoltat numeroase limbaje de nivel înalt. Dintre acestea, unele au rezistat în timp și au fost utilizate pe scară largă : FORTRAN (limbaj de uz general, în special pentru oameni de știință și ingineri, care în timp a avut numeroase versiuni), COBOL (limbaj proiectat pentru aplicații economice), LISP (care a introdus concepțele programării funcționale) și ALGOL (care nu a avut importanță comercială, dar a introdus concepție-cheie pentru dezvoltarea limbajelor de programare actuale).

Perioada următoare a fost extrem de prolifică, atât în ceea ce privește limbajele de programare dezvoltate, cât și în privința concepțiilor care au influențat evoluția limbajelor de programare. Un prim concept-cheie a fost notația BNF (*Backus-Naur-Form*), introdusă în 1959, reprezentând o modalitate matematică de a descrie riguros sintaxa unui limbaj de programare.

Principala paradigmă de programare dezvoltată în această perioadă a fost programarea structurată, ca răspuns la necesitatea de a scrie programe mai clare, în timp mai scurt și a căror corectitudine să poată fi demonstrată. Statisticile demonstравă că la vremea respectivă un programator scria în medie între 5 și 10 linii de cod corecte pe zi. Extrem de puțin, având în vedere dezvoltarea *hardware*-ului și cerința pentru *software* existentă. Punctul de plecare al programării structurate este considerat articolul „Go To Statement Considered Harmful”, scris în 1968 de Edsger Dijkstra, precum și lucrările lui Böhm și Jacopini. Aceștia au formulat și demonstrat în 1966 Teorema programării structurate : „orice program poate fi descris cu ajutorul a trei structuri de control : structura secvențială, structura alternativă și structura repetitivă”. Timp de 20 de ani, Dijkstra a promovat programarea structurată, dar probabil momentul de cotitură, prin care industria *software* a conștientizat beneficiile acestei paradigmă de programare, a fost lansarea, în 1972, a unei aplicații de indexare a articolelor din ziare, realizată de IBM pentru *New York Times*. Dezvoltarea aplicației a durat 22 de luni, echivalând cu munca unui om în 11 ani, constând în aproximativ 83.000 de linii de cod. După cinci săptămâni de testare a aplicației au fost identificate doar 21 de erori, urmând ca în următorul an de utilizare să mai fie identificate încă 25.

Mai mult, aplicația a fost livrată înainte de termen și cheltuielile au fost mai mici decât cele bugetate. Acest succes a fost un soc în comunitatea programatorilor (de exemplu, aplicația precedentă dezvoltată de IBM, sistemul de operare pentru System/360, a costat sute de milioane de dolari, a fost finalizat cu un an întârziere și conținea la lansare mii de erori). Programarea structurată a constituit o „revoluție” în domeniul.

În esență, programarea structurată are la bază trei principii pentru dezvoltarea aplicațiilor :

1. *analiza TOP-DOWN*: problema de rezolvat este împărțită în subprobleme relativ independente ; fiecare subproblemă este analizată în același mod ;
2. *modularizarea* : programul este împărțit în mai multe secțiuni (denumite module, subprograme, proceduri sau funcții), fiecare modul rezolvând o subproblemă a problemei date (în concordanță cu analiza TOP-DOWN a problemei) ;
3. *codul structurat* : fiecare modul constă în compunerea celor trei structuri de control fundamentale : secvențială, alternativă, repetitivă.

Interesant este că principiile care stau la baza programării orientate pe obiecte datează tot din anii '60. Cercetătorii de la Universitatea din Oslo au dezvoltat un limbaj de programare denumit Simula, care introducea noțiunile de *clasa*, *obiect*, *funcție virtuală* etc. Simula fiind primul limbaj de programare orientată pe obiecte. Inspirati de acest limbaj, cercetătorii de la XEROX Palo Alto au dezvoltat în perioada 1970-1980 un alt limbaj de programare orientată pe obiecte „pur”, SmallTalk, în care toate prelucrările se realizează în cadrul claselor. O încercare de a crea un limbaj de programare orientată pe obiecte „pur”, eficient de utilizat în industria software, a fost limbajul Eiffel, creat de Bertrand Meyer. Programarea orientată pe obiecte a devenit însă populară odată cu apariția limbajului C++. Limbajul C++ creat la începutul anilor '80 de Bjarne Stroustrup de la AT&T Labs era un limbaj hibrid, care păstra compatibilitatea cu limbajul C. Limbajul este considerat hibrid deoarece programatorul poate scrie un cod și în afara claselor. Ulterior, la începutul anilor '90, Sun Microsystems a dezvoltat un alt limbaj de programare orientată pe obiecte, denumit Java. Java a menținut o sintaxă asemănătoare cu a limbajului C++, dar a eliminat caracteristici potențial periculoase (cum ar fi posibilitatea de a utiliza pointeri) și a simplificat anumite aspecte (de exemplu, are un sistem denumit *garbage collection*, care eliberează automat memoria neutilizată).

La ora actuală, programarea orientată pe obiecte este paradigma general acceptată pentru dezvoltarea de aplicații *software*.

1.2. Principiile POO

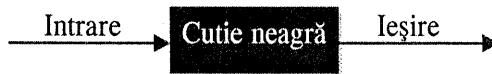
Abstractizarea datelor

Un *tip de date* este definit de domeniul de valori pe care le pot lua datele de tipul respectiv, setul de operații permise asupra valorilor respective și modul de reprezentare în memorie.

Un *tip de date abstract* este definit doar de domeniul de valori și setul de operații asupra valorilor respective, modul de reprezentare în memorie nefiind specificat pentru utilizator.

De exemplu, stiva, ca tip de date abstract, este o structură de date care permite două operații: *push* (inserarea unui element la vârful stivei) și *pop* (extragerea elementului din vârful stivei). Modul de implementare a stivei (Ca vector? Ca listă simplu înlănțuită?) nu este specificat, ci doar setul de operații permise.

Un tip de date abstract practic funcționează ca o „cutie neagră”: interiorul este „opac”, poate fi studiat/utilizat pe bază de intrări/ieșiri.



Abstractizarea datelor este procesul prin care sunt definite de către utilizator tipuri de date abstrakte (ADT = *Abstract Data Types*). În C++, tipurile de date abstrakte se implementează utilizând *clase*.

Încapsularea datelor

Încapsularea datelor (*encapsulation*, în limba engleză) este o consecință imediată a principiului abstractizării datelor, proces în care reprezentarea internă este „ascunsă” utilizatorului.

Procesul de dezvoltare a aplicațiilor până la *POO* poate fi descris în modul cel mai concis prin célébra ecuație lansată în 1976 de Niklaus Wirth:

$$\text{Structuri de date} + \text{Algoritm} = \text{Program}$$

Această ecuație evidențiază faptul că efortul de proiectare a unei aplicații se concentrează pe o bună structurare a datelor, apoi pe dezvoltarea celor mai eficienți algoritmi de prelucrare a acestora.

Efortul de proiectare a unei aplicații *POO* constă în primul rând în identificarea „modulelor” aplicației, datele fiind „ascunse” în cadrul fiecărui modul. Modulul trebuie să conțină funcții „de interfață”, care să permită accesul la datele „ascunse” în modul. Utilizatorul modulului nu are acces direct la date, acestea fiind permis doar prin intermediul funcțiilor de „interfață”. Chiar dacă reprezentarea datelor se schimbă și, în consecință, și implementarea funcțiilor de prelucrare a datelor, acest lucru nu este transparent pentru utilizator, fiindcă accesul se realizează prin aceeași interfață. În plus, accesul la date fiind limitat, crește securitatea aplicației.

Moștenirea

Conceptul de moștenire (*inheritance*, în limba engleză) este inspirat din domeniul biologiei. De exemplu, putem considera ca tip de date abstract clasa *Animal*. Toate animalele trebuie să aibă funcție de hrănire, funcție de reproducere etc. Acestea sunt funcții comune tuturor animalelor. În cadrul clasei *Animal* putem diferenția diferite tipuri de animale, de exemplu, păsări și mamifere. Atât păsările, cât și mamiferele au în comun elementele specifice animalelor, dar au și caracteristici specifice. De exemplu, păsările au pene.

La *POO*, prin moștenire, o clasă poate moșteni toate elementele altelor clase. Procesul prin care o clasă moștenește elementele altelor clase mai este cunoscut și sub denumirea de *derivare*. Prin derivare, o clasă (denumită clasă derivată sau subclasa) moștenește toate elementele unei alte clase, cea din care este derivată (aceasta se numește clasă de bază sau supraclasă). Clasa derivată va avea și alte elemente specifice ei, care o diferențiază de clasa de bază. De exemplu, clasa *Pasare* poate fi derivată din clasa *Animal* și adăugăm la caracteristici specifice faptul că păsările au pene. Clasa derivată este mai specializată decât clasa de bază.

O clasă derivată poate fi, la rândul ei, clasă de bază pentru o altă clasă. Se obțin astfel ierarhii de clase, clasele derivează având în comun toate elementele moștenite de la strămoșul lor comun.

În concluzie, în procesul de proiectare a unei aplicații *POO*, trebuie să identificăm „modulele” aplicației (clasele), dar și elementele pe care le au acestea în comun, construind, prin derivare, ierarhii de clase. Prin moștenire eliminăm codul redundant și putem extinde aplicația, adăugând noi funcționalități.

Polimorfismul

Etimologic, cuvântul polimorfism provine din limba greacă, de la *polys* (multe) și *morphos* (formă). În *POO*, prin *polimorfism* se înțelege posibilitatea ca, prin apelarea unei funcții, să obținem efecte diferite, în funcție de contextul apelului. Reluând analogia cu lumea vie, menționam că toate animalele trebuie să aibă funcția de reproducere; dar această funcție „se execută” diferit în funcție de context (într-un mod are loc, de exemplu, reproducerea la păsări, în alt mod la mamifere).

În C++, polimorfismul se poate obține prin supraîncărcarea funcțiilor (*overloading*) sau, în cazul claselor derivează, prin redefinirea unei funcții a clasei de bază (*overriding*).

1.3. Avantajele *POO*

Principiile care stau la baza *POO* conduc la o serie de avantaje în ceea ce privește dezvoltarea aplicațiilor. Aplicațiile au o structură modulară. Modulele:

- pot fi dezvoltate separat;
- pot fi modificate separat (fără a afecta funcționalitatea întregii aplicații);
- pot fi reutilizate în alte aplicații;
- pot fi extinse ulterior.

Toate acestea conduc la creșterea productivității (timpul necesar pentru dezvoltare este mai scurt, costurile sunt reduse, mențenanța aplicației este mai ușoară).

1.4. Întrebări recapitulative

1. Care dintre următoarele afirmații sunt corecte ?
 - a. Limbajul C++ permite moștenirea, dar aceasta nu este recomandată, deoarece este consumatoare de timp.
 - b. Prințipiu încapsulării datelor presupune ca datele să fie toate împreună în aceeași structură de date, fiind permis accesul la datele din structură.
 - c. Abstractizarea datelor înseamnă să definim sumar structura de date.
 - d. În limbajele care permit programarea orientată pe obiecte nu se mai utilizează programarea structurată.
 - e. Niciuna dintre variantele precedente.
2. Prințipiu care permite ca aceeași acțiune să fie executată în mod diferit în funcție de context se numește :
 - a. poligamie ;
 - b. moștenire ;
 - c. polimorfism ;
 - d. *multitasking*.
3. Dați exemple din viața cotidiană care ilustrează moștenirea.
4. Dați exemple din viața cotidiană care ilustrează polimorfismul.

u dezvoltare
ă).

tă, deoarece

împreună în
ră.
e.
ai utilizează

it în funcție

2. Programarea orientată pe obiecte în C++

2.1. Clasele și obiectele

În dicționar, termenul *clasă* este definit ca o mulțime de elemente cu însușiri comune. Prin însușiri comune putem înțelege atribute comune sau comportamente comune. Să considerăm clasa câinilor. Toți câinii au atribute și comportamente comune. De exemplu, toți câinii au blană, deci un atribut interesant ar putea fi lungimea părului. Toți câinii latră – acesta este un comportament comun.

În esență, o clasă reprezintă exact același lucru și în cazul *POO*. Clasa definește caracteristicile comune (attribute, comportamente) ale obiectelor clasei. Mai exact, cu ajutorul claselor putem defini propriile noastre tipuri de date.

Din punct de vedere sintactic, declarația unei clase este :

```
class NumeClasă  
{listă_declarații_membri};
```

unde :

- NumeClasă = numele tipului clasă declarat
- listă_declarații_membri = secvență de declarații ale membrilor clasei (conține declarații de date membre, precum și declarații sau definiții de funcții membre).

Datele membre corespund atributelor, iar funcțiile membre corespund comportamentelor. Funcțiile membre ale unei clase mai sunt denumite *metode*.

Ca prim exemplu, să declarăm clasa Câine :

```
class Câine  
{  
    int LungimePăr;           //dată membru  
    void Latră() {cout<<"Ham!";} //funcție membru  
};
```

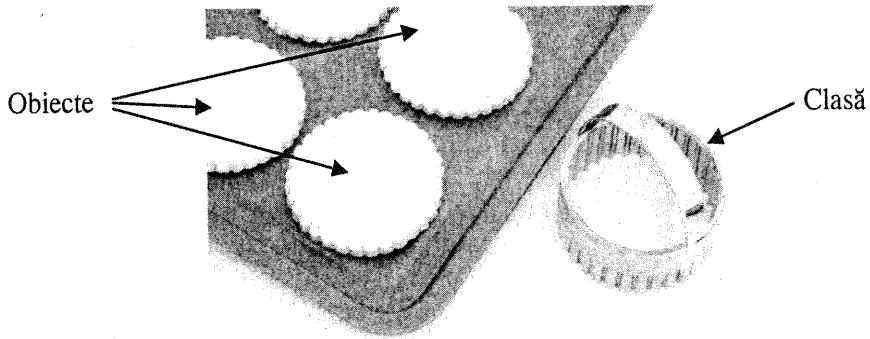
Un obiect este o variabilă de tip clasă sau, mai elegant formulat, o instanțiere a unei clase.

De exemplu :

```
Câine Grivei, Azorel;
```

Grivei și Azorel sunt două instanțe ale clasei Câine (două obiecte).

O analogie relevantă pentru a înțelege mai bine conceptele de *clasă* și *obiect* este cea cu forma de prăjituri. Forma de prăjituri este clasa ; ea definește aspectul tuturor prăjiturilor care vor fi tăiate cu forma respectivă. Prăjiturile pe care le tăiem cu forma respectivă reprezintă obiectele clasei.



Accesul la un membru al unui obiect se realizează în mod similar cu accesul la un câmp al unei variabile de tip struct, utilizând operatorul de selecție directă :

| **obiect.membru**

De exemplu :

```
| Grivei.LungimePăr=10;
| //pentru obiectul Grivei am atribuit datei membru
| //LungimePăr valoarea 10
| Grivei.Latră();
| //am apelat funcția membru Latră() pentru obiectul Grivei
```

2.2. Controlul accesului la membrii unei clase

În procesul de proiectare a unei clase, o atenție specială trebuie acordată controlului modului de acces pentru membrii unei clase. În acest scop pot fi utilizati trei specificatori de control al accesului :

- **public** = membrul poate fi accesat din orice funcție din domeniul de declarație al clasei ;
- **private** = membrul poate fi accesat doar din funcțiile membre ale clasei, precum și din funcțiile prietene (friend) ale clasei ;
- **protected** = ca efect, este similar cu private, dar, în plus, accesul este permis și pentru funcțiile membre și funcțiile prietene ale claselor derivate din clasa respectivă.

Specificatorul implicit este **private**.

Din punct de vedere sintactic, un specificator de control al accesului se precizează astfel :

| **specificator:**

Observații

1. Un specificator de acces rămâne valabil până la apariția unui alt specificator.
2. Conform principiului încapsulării datelor, datele membre ale unei clase ar trebui să fie „ascunse” în interiorul clasei. Cu alte cuvinte, nu trebuie să fie publice. Clasa ar trebui să conțină metode speciale publice, prin care putem să aflăm valoarea datei membre sau să o modificăm. Aceste funcții se numesc accesori. Este frecvent utilizat prefixul **Get** pentru numele unui accesator care returnează valoarea, respectiv prefixul **Set** pentru numele unui accesator care modifică valoarea datei membre.
3. În general, este de dorit să minimizăm numărul de membri publici ai unei clase; membrii publici vor constitui partea de „interfață”, prin intermediul cărei clasa interacționează cu exteriorul.
4. O analogie relevantă pentru înțelegerea modului de proiectare și de utilizare a unei clase este cea cu aparatul de radio. Un utilizator obișnuit al unui aparat de radio obține efectele dorite acționând butoanele de pe carcasa aparatului de radio (pentru a deschide aparatul, apasă pe un buton, pentru a modifica volumul, rotește un alt buton etc.). Butoanele acestea reprezintă de fapt metodele publice ale clasei. În mod normal, un utilizator al aparatului de radio nu va desface carcasa aparatului pentru a avea acces direct la ceea ce este în interior (interiorul aparatului reprezentând membrii privați ai clasei).

Exemplu

Să rescriem clasa **Câine**, controlând accesul la data membră **LungimePăr**:

```
class Câine
{
private:
    int LungimePăr;
public:
    void Latră() { cout << "Ham! " ; }
    int GetLungimePăr() { return LungimePăr; }
    int SetLungimePăr(int x) { LungimePăr = x; }
};
```

Data membră **LungimePăr** este privată, în timp de metodele **Latră()**, **GetLungimePăr()** și **SetLungimePăr()** sunt publice (ultimele două fiind accesori).

Pentru a stabili lungimea părului lui Grivei, vom apela accesatorul **SetLungimePăr()**:

```
Grivei.SetLungimePăr(10);
```

Dacă dorim să afișăm lungimea părului lui Grivei:

```
cout << Grivei.GetLungimePăr();
```

O încercare de a accesa direct data membră **LungimePăr()** sub forma:

```
cout << Grivei.LungimePăr;
```

va genera un mesaj de eroare la compilare.

2.3. Arhitectura unei aplicații *POO*

Pentru fiecare clasă se generează două fișiere :

- un fișier care conține declarația clasei ; acesta va fi un fișier antet (*header*) , care va avea numele clasei și extensia .h ;
- un fișier care conține implementarea clasei ; în acest fișier sursă (fișier cu numele clasei și cu extensia .cpp) se află definițiile funcțiilor clasei.

În orice fișier sursă este necesar să utilizăm clasa și vom include fișierul antet :

```
| #include "NumeClasa.h"
```

O aplicație va consta dintr-un proiect care va conține mai multe fișiere sursă, doar unul dintre acestea conținând funcția main(). Orice funcție va fi definită o singură dată (în fișierul sursă în care apare implementarea sa), dar va fi declarată ori de câte ori este necesar (prin includerea fișierului antet corespunzător).

2.4. Definirea funcțiilor membre în exteriorul clasei

Când definim o funcție membră a unei clase în exteriorul clasei trebuie să precizăm și clasa căreia îi aparține. Pentru aceasta vom preceda numele funcției de numele clasei, utilizând operatorul de rezoluție :: astfel :

```
| NumeClasă::NumeFuncțieMembră
```

Este destul de firesc, având în vedere că în clase diferite pot exista funcții cu același nume.

Exemplu

Vom începe să construim pas cu pas o clasă pentru lucrul cu fracții. Pentru a specifica o fractie trebuie să reținem numitorul și numărătorul fractiei (acestea vor fi datele membre private). Pentru început, vom defini accesori și o funcție de simplificare. Declarația clasei se va afla în fișierul antet fractie.h :

```
class fractie
{
private:
    int a; //numarator
    int b; //numitor
public:
    int GetNumarator() { return a; }
    int GetNumitor() { return b; }
    void SetNumitor(int);
    void SetNumarator(int);
    void Simplifica();
};
```

Observați că pe două dintre funcțiile membre (`GetNumarator()` și `GetNumitor()`) le-am definit în cadrul declarației clasei. Celelalte trei funcții le vom defini în exteriorul clasei în fișierul `fractie.cpp`:

```
#include "fractie.h"

void fractie::SetNumarator(int x)
{ a=x; }

void fractie::SetNumitor(int y)
{ b=y; }

void fractie::Simplifica()
{int x, y, r;
 //calculez cmmdc cu algoritmul lui Euclid
 x=a; y=b;
 while (y) { r=x%y; x=y; y=r; }
 a/=x; b/=x; //simplific
}
```

Ca să ilustrăm modul de utilizare a clasei `fractie`, în funcția `main()` vom declara un obiect de tip `fractie`, vom inițializa numitorul și numărătorul, vom simplifica fracția, apoi o vom afișa simplificată.

```
#include <iostream>
#include "fractie.h"

using namespace std;

int main()
{
    fractie f;
    f.SetNumitor(12);
    f.SetNumarator(8);
    f.Simplifica();
    cout<<f.GetNumarator()<<'/'<<f.GetNumitor();
    return 0;
}
```

Evident, executând acest program, se va afișa pe ecran `2/3`

2.5. Funcțiile *inline*

Diferența dintre o funcție *inline* și una uzuale constă în faptul că fiecare apel al funcției *inline* este înlocuit de compilator cu codul funcției.

Avantajul constă în creșterea vitezei de execuție a codului, fiind evitate operațiile aferente apelului de funcție (de exemplu, alocarea pe stivă a memoriei necesare apelului, copierea valorilor parametrilor, returnarea valorii calculate de funcție). Dezavantajul constă în faptul că va crește dimensiunea codului sursă.

Din punct de vedere sintactic, definiția unei funcții *inline* este similară cu definiția unei funcții uzuale, numai că este precedată de cuvântul-cheie *inline*.

Exemplu

```
inline int max(int a, int b)
{ return a > b ? a : b; }
```

Observații

1. Funcțiile *inline* trebuie să fie foarte scurte.
2. Funcțiile recursive nu pot fi funcții *inline*.
3. Compilatorul poate ignora declarația *inline*.
4. Funcțiile *inline* combină avantajele macrourilor cu cele ale funcțiilor. În acest mod, ca și în cazul macrourilor, se elimină operațiile aferente apelului, dar păstrează toate proprietățile funcțiilor în ceea ce privește validitatea apelului, transferul parametrilor, declarațiile.

Funcțiile membre definite în declarația clasei sunt implicit funcții *inline*.

În exemplul precedent, funcțiile `GetNumitor()` și `Getnumarator()` sunt, prin urmare, funcții *inline*.

2.6. Funcțiile cu parametri implicați

Pentru a corespunde unei varietăți cât mai mari de situații, funcțiile sunt proiectate de obicei cu un număr foarte mare de parametri, chiar dacă pentru unii parametri se utilizează frecvent la apel aceeași valoare.

Pentru ca funcțiile respective să fie mai ușor de utilizat, limbajul *C++* a introdus o facilitate suplimentară, prin care programatorul poate declara în antetul funcției valori implicate.

Exemplu

```
void f(int a, int b=1, int c=2)
{ cout << a << b << c; }
```

Funcția `f()` are trei parametri, dintre care doi cu valori implicate (`b` are valoarea implicită 1, iar `c` are valoarea implicită 2).

La apelul funcției se pot omite parametrii actuali corespunzători parametrilor formali cu valori implicate, în acest caz utilizându-se valorile implicate ale acestora. Pentru ca la apel corespondența parametru actual – parametru formal să se execute corect, parametrii cu valori implicate sunt întotdeauna ultimii declarați în lista de parametri a funcției.

Valorile implicate se specifică o singură dată (fie în definiția funcției, fie în declarația acesteia; ușual sunt specificați în declarație, deoarece compilatorul validează apelul de funcție în conformitate cu declarația acesteia).

ă cu definiția

Exemplu

Funcția precedentă f () definită mai sus o putem apela cu un parametru :

```
f(9);
```

În acest caz se utilizează valorile implicate ale parametrilor b și c și se afișează 912.

Sau o putem apela cu doi parametri :

```
f(9, 8);
```

În acest caz se utilizează valoarea implicită a parametrului c și se afișează 982.

Sau o putem apela cu trei parametri :

```
f(9, 8, 7);
```

În acest caz nu se utilizează valorile implicate ale parametrilor și se afișează 987.

Exemplu

Un exemplu de funcție cu parametri implicați pe care l-ați utilizat deja este funcția getline (), funcție membră a clasei istream:

Prototip :

```
istream & getline(signed char*, int, char = '\n');
```

Funcția citește caracterele de la intrare și le plasează în sirul descris de primul parametru. Operația se termină fie când s-au extras atâtea caractere câte specifică cel de-al doilea parametru, fie la întâlnirea marcajului de sfârșit, specificat de cel de-al treilea parametru. Observați că cel de-al treilea parametru are ca valoare implicită caracterul *newline* (deci, implicit, citirea se termină la sfârșitul liniei).

2.7. Supraîncărcarea funcțiilor

Cea mai puternică facilitate introdusă de limbajul C++ referitor la funcții constă în posibilitatea de supraîncărcare¹ a funcțiilor.

Supraîncărcarea oferă programatorului posibilitatea de a defini mai multe funcții cu același nume, dar care diferă prin numărul și/sau tipul parametrilor.

Exemplu

Definim o funcție suma () care calculează suma a două numere întregi :

```
int suma (int a, int b)
{return a+b;}
```

1. Termenul „supraîncărcare” este o traducere a termenului englezesc *overloading*. În unele cărți puteți întâlni și termenul „supradefinire”, cu aceeași semnificație.

Putem supraîncărca funcția suma(), astfel încât să calculeze suma elementelor unui vector :

```
int suma (int * a, int n)
{
    for (int i=0, s=0; i<n; i++) s+=a[i];
    return s;
}
```

Deși nu am discutat explicit despre aceasta, am folosit deja supraîncărcarea funcțiilor. De exemplu, funcția get(), funcție membru a clasei istream, este o funcție supraîncărcată, având următoarele prototipuri :

```
Forma 1: int get();
Forma 2: istream& get(char &);
Forma 3: istream& get(char *, int lg, char = '\n');
Forma 4: istream& get(streambuf &, char = '\n');
```

Compilatorul este cel care selectează la apel funcția adecvată în funcție de lista parametrilor actuali. În primul rând, se caută o corespondență exactă (o funcție pentru care tipurile parametrilor formali să coincidă cu parametrii actuali ai apelului respectiv).

Dacă nu este găsită o corespondență exactă, se fac conversii de tip implicate (de exemplu, tipul float este convertit la double; char este convertit la int etc.). În acest caz pot apărea ambiguități (compilatorul nu poate identifica în mod unic funcția pe care trebuie să o apeleze).

Exemplu

Probabil cel mai frecvent caz de ambiguitate apare din cauza funcției sqrt(). În C++ funcția sqrt() este supraîncărcată astfel :

```
long double sqrt(long double);
float sqrt(float);
double sqrt(double);
```

Un apel de tipul :

```
int x;
... sqrt(x) ...
```

va genera la compilare un mesaj de eroare, deoarece compilatorul nu știe la care dintre cele trei tipuri reale să-l convertească pe x.

Utilizarea funcțiilor supraîncărcate reprezintă unul dintre cele mai importante mecanisme ale programării orientate obiect.

2.8. Constructorii

Constructorii sunt funcții membre ale clasei care au caracteristici speciale :

- se apelează automat la crearea fiecărui obiect al clasei ;

- numele constructorilor coincide cu numele clasei;
- în declarația/definiția unui constructor NU se specifică tipul rezultatului.

O clasă poate avea mai mulți constructori, prin supraîncărcare. Dacă în cadrul declarației clasei nu am specificat niciun constructor, compilatorul va genera automat un constructor implicit (constructor fără parametri).

Pentru a putea crea obiecte ale clasei, constructorii trebuie să fie publici.

Exemplu

Vom completa declarația clasei fractie cu un constructor, astfel:

```
class fractie
{
private:
    int a; //numarator
    int b; //numitor
public:
    int GetNumarator(){ return a; }
    int GetNumitor() { return b; }
    void SetNumitor(int);
    void SetNumarator(int);
    void Simplifica();
    fractie(int x=0, int y=1);
};
```

Vom defini acest constructor în exteriorul clasei astfel:

```
fractie::fractie(int x, int y)
{ a=x; b=y; }
```

Observați că am ales să definim un singur constructor, cu valori implicite (valorile implicite le-am specificat la declararea constructorului). Pentru exemplificare, în funcția main() am creat trei obiecte ale clasei (f1, f2, f3), în trei moduri diferite:

- folosind constructorul fără parametri pentru obiectul f1 (în acest caz s-au folosit valorile implicite ale parametrilor, adică f1 reprezintă fracția 0/1);
- folosind constructorul cu un singur parametru pentru obiectul f2 (în acest caz pentru al doilea parametru s-a folosit valoarea implicită, deci f2 reprezintă fracția 5/1);
- folosind constructorul cu doi parametri pentru obiectul f3 (f3 reprezintă fracția 8/12).

```
#include <iostream>
#include "fractie.h"
using namespace std;
int main()
{
    fractie f1, f2(5), f3(8,12);

    cout<<f1.GetNumarator()<<'/'<<f1.GetNumitor()<<'\n';
    cout<<f2.GetNumarator()<<'/'<<f2.GetNumitor()<<'\n';
    cout<<f3.GetNumarator()<<'/'<<f3.GetNumitor()<<'\n';
```

```

f3.Simplifica();

cout<<f3.GetNumarator()<<'/'<<f3.GetNumitor()<<'\n';
return 0;
}

```

Evident, executând acest program, obținem pe ecran:

```

0/1
5/1
8/12
2/3

```

Exemplu

Clasa `fractie` putea fi utilizată și fără a declara un constructor. S-ar fi generat automat un constructor implicit și am fi putut utiliza clasa în modul prezentat la prima formă de declarare a clasei.

Pentru a exemplifica și o situație în care declararea unui constructor este obligatorie, vom declara și vom implementa o clasă denumită `mesaj`. Datele membru sunt lungimea mesajului și un vector care va conține caracterele mesajului. Pentru ca memoria alocată vectorului să fie exact cea necesară, vom aloca dinamic memorie pentru mesaj. În concluzie, la crearea unui obiect al clasei trebuie să alocăm dinamic memoria necesară (evident, în constructori).

Declarația clasei `mesaj` (stocată în fișierul `mesaj.h`) va arăta astfel:

```

class mesaj
{
private:
    char * sir;
    int lg;
public:
    mesaj();           //constructor fara parametri
    mesaj (char *);  //constructor supraincarcat
    int GetLg();
    void Scrie();
...
};

```

Declarația este incompletă, urmând să o completăm în secțiunile următoare. Observați că am declarat doi constructori: un constructor fără parametri și unul care primește ca parametru un sir de caractere.

În fișierul sursă `mesaj.cpp` vom defini constructorii și funcțiile membre `GetLg()` și `Scrie()` astfel:

```

mesaj::mesaj()
{ lg=0; sir=NULL; }

mesaj::mesaj (char * s)
{int i;
lg=strlen(s); //determin lungimea sirului

```

```

\n';
sir= new char[lg];
//aloc dinamic memorie pentru vectorul care retine
//caracterele mesajului
for (i=0; i<lg; i++) sir[i]=s[i];
//copiez caracterele mesajului
}

int mesaj::GetLg()
{ return lg; }

void mesaj::Scrie()
{ if (sir)
    for (int i=0; i<lg; i++) cout<<sir[i];
    else
        cout<<"Mesajul este vid";
    cout<<'\n';
}

```

S-ar fi generat un obiect de tip mesaj și prezentat la ecran. Constructorul este obiectul membru al clasei. Pentru că amicul memorie locăm dinamic în memoria dinamică.

1:

```

#include <iostream>
#include "mesaj.h"
using namespace std;
int main()
{
    mesaj m1, m2("Ana are mere.");
    cout<<m1.GetLg()<<' '; m1.Scrie();
    cout<<m2.GetLg()<<' '; m2.Scrie();
    return 0;
}

```

Executând acest program, se afișează pe ecran :

```

0 Mesajul este vid
13 Ana are mere.

```

ile următoare. tri și unul care

nbre GetLg()

2.9. Destructorul

Destructorul este de asemenea o funcție membră specială a unei clase, datorită caracteristicilor sale :

- este apelat automat la eliminarea unui obiect al clasei (la încheierea ciclului de viață al obiectului sau, în cazul obiectelor dinamice, atunci când este utilizat operatorul delete);
- trebuie să existe; dacă în cadrul clasei nu este explicit declarat un destructor, compilatorul generează automat un destructor implicit;
- destructorul nu poate fi supraîncărcat (este unic);

- are același nume ca și clasa, precedat de caracterul ~ ;
- nu are argumente și nici tip rezultat.

Exemplu

În clasa fractie nu vom declara un destructor, pentru că nu este necesar (compilatorul va genera automat un destructor). Dar, în clasa mesaj, un destructor este obligatoriu, pentru că atunci când un obiect al clasei este eliminat, trebuie să fie eliberată și memoria alocată dinamic pentru vectorul de caractere.

Declarația destructorului în cadrul declarației clasei va fi :

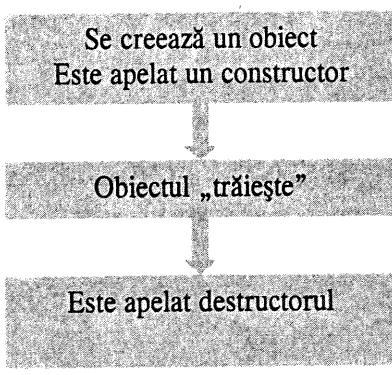
```
| ~mesaj();
```

Iar definiția destructorului în exteriorul clasei va fi :

```
| mesaj::~mesaj()
| {
|     delete sir; //eliberez memoria alocata
| }
```

2.10. Modelul logic al vieții unui obiect

Modelul logic al vieții unui obiect poate fi ilustrat astfel :



Exemplu

În exemplul următor am declarat o clasă denumită fictiv, în care am inclus un constructor, un destructor și funcția membră Scrie(). Toate cele trei funcții membre nu fac decât să afișeze pe ecran câte un mesaj care are rolul de a evidenția momentul execuției funcției.

Fisierul fictiv.h:

```
| class fictiv
| {
|     public:
|         fictiv();
```

```

    ~fictiv();
    void Scrie();
};


```

Fișierul fictiv.cpp:

```

#include "fictiv.h"
#include <iostream>
using namespace std;
fictiv::fictiv() //constructor
{ cout<<"Am creat un obiect!\n"; }
fictiv::~fictiv() //destructor
{ cout<<"Am eliminat un obiect!\n"; }
void fictiv::Scrie()
{ cout<<"Obiectul traieste!\n"; }


```

În fișierul main.cpp ilustrăm ciclul de viață al unui obiect în două moduri:

- în funcția f() am declarat o variabilă locală ob, de tip fictiv; acest obiect va fi creat în momentul apelării funcției f(), va „trăi” pe durata execuției funcției f() și va fi eliminat la terminarea funcției f();
- în funcția main() am declarat un pointer la un obiect de tip fictiv denumit p_ob, cu scopul de a reține adresa unui obiect alocat dinamic; la alocarea dinamică a obiectului se apelează automat constructorul; acest obiect va „trăi” până la eliberarea sa dinamică, folosind operatorul delete.

```

#include <iostream>
#include "fictiv.h"
using namespace std;
void f()
{fictiv ob;
 cout<<"Suntem in functia f: "; ob.Scrie(); }
int main()
{ f();
 fictiv * p_ob=new fictiv;
 cout<<"Suntem in functia main: "; p_ob->Scrie();
 delete p_ob;
 return 0; }


```

Executând acest program, pe ecran se va afișa:

```

Am creat un obiect!
Suntem in functia f: Obiectul traieste!
Am eliminat un obiect!
Am creat un obiect!
Suntem in functia main: Obiectul traieste!
Am eliminat un obiect!


```

Observație

În funcția main(), pentru a accesa funcția membru Scrie() a obiectului alocat dinamic, am utilizat operatorul de selecție indirectă ->, întrucât am accesat obiectul

este necesar
un destritor
trebuie să fie

am inclus un
le trei funcții
de a evidenția

indirect, prin intermediul pointerului care conține adresa sa. Reamintim că efectul construcției `p->membru` este echivalent cu `(*p).membru`.

2.11. Constructorul de copiere

O altă funcție membră specială este constructorul de copiere. Constructorul de copiere se apelează automat ori de câte ori este necesară copierea unui obiect al clasei respective (de exemplu, la transferul unui obiect ca parametru sau rezultat al unei funcții, la crearea unor obiecte temporare).

Orice clasă trebuie să aibă un constructor de copiere. Dacă nu este declarat explicit unul, compilatorul va genera automat un constructor de copiere implicit, care copiază obiectul membru cu membru.

Numele constructorului de copiere coincide cu numele clasei, nu are tip rezultat, iar ca parametru trebuie să primească o referință la un obiect al clasei respective.

| numeclasa (numeclasa & obiect);

Exemplu

Să revenim la clasa `mesaj`, pe care o vom completa cu un constructor de copiere. Declarația constructorului din cadrul clasei `mesaj` va fi:

| mesaj (mesaj&);

Definiția constructorului de copiere din exteriorul clasei va fi:

```
mesaj::mesaj (mesaj& m)
{
    int i;
    lg=m.lg;           //copiez lungimea
    sir=new char[lg]; //aloc memorie pentru vector
    for (i=0; i<lg; i++) sir[i]=m.sir[i];
    //copiez caracterele din mesajul m in mesajul curent
}
```

Dacă nu am fi declarat un constructor de copiere în clasa `mesaj`, compilatorul ar fi generat un constructor de copiere. Dar acest constructor nu ar fi funcționat corect în acest caz, deoarece unul dintre membrii clasei este un pointer, iar noi nu intenționăm să copiem pointerul, ci conținutul zonei de memorie a cărei adresă se află în pointerul respectiv. Ca urmare a copierii pointerului vor apărea două probleme:

- dacă intenționăm să modificăm zona de memorie a cărei adresă se află în pointer pentru obiectul copie, va fi afectată de fapt zona de memorie indicată de pointerul din obiectul copiat (sunt doi pointeri diferiți, dar care indică aceeași zonă de memorie);
- când obiectul copie este eliminat (el fiind un obiect temporar) este apelat destructorul; destructorul va încerca să elibereze zona de memorie alocată dinamic pentru obiectul copie; pointerul copie și pointerul din obiectul copiat indicând o aceeași zonă de memorie, se eliberează o zonă de memorie care nu ar trebui eliberată.

În concluzie, atunci când membrii unei clase sunt pointeri către zone de memorie ce urmează să fie alocate dinamic, constructorul de copiere generat automat de compilator nu este suficient, trebuie să definim explicit un constructor de copiere.

2.12. Pointerul `this`

În fiecare funcție membră nestatică a unei clase este automat declarat un pointer constant, denumit `this`, care are ca valoare adresa obiectului curent.

Uneori este necesar să utilizăm și noi explicit pointerul `this`. De exemplu, atunci când o funcție trebuie să returneze ca rezultat obiectul însuși :

```
| return *this;
```

2.13. Membrii statici ai unei clase

O clasă poate conține membri statici (date și funcții).

Din punct de vedere sintactic, specific este faptul că declararea membrilor statici este precedată de cuvântul-cheie `static`.

Referirea la un membru static al unei clase din exteriorul clasei se poate realiza în două moduri :

```
| NumeObiect.NumeMembruStatic  
NumeClasa::NumeMembruStatic
```

Cel de-al doilea mod de referire este posibil datorită faptului că membrii statici sunt aceiași pentru toate obiectele clasei (deci nu caracterizează un anumit obiect, ci întreaga clasă).

Datele membre statice

Pentru datele membre nestatice există copii distincte în fiecare obiect al clasei. Datele membre statice au o valoare unică, comună pentru toate obiectele clasei. Din acest motiv, datele membre statice sunt supranumite *variabilele clasei*, deoarece se comportă ca variabilele globale.

Membrii statici reprezintă o soluție optimă în cazul în care ansamblul obiectelor clasei utilizează date în comun, deoarece :

- se reduce numărul de variabile globale ;
- se asociază explicit datele statice cu o anumită clasă ;
- spre deosebire de variabilele globale, în cazul membrilor statici este posibil controlul accesului.

Deoarece pentru o dată membră statică se asociază o singură zonă de memorie pentru întreaga clasă, trebuie să existe o definiție a datei membre statice în afara clasei. Sintaxa unei astfel de definiții este :

```
| tip NumeClasa::NumeMembruStatic;
```

La definire este posibilă și inițializarea datei membre (implicit compilatorul va inițializa cu 0 datele membre statice).

Exemplu

Pentru exemplificarea modului de funcționare a unei date membre statice, vom declara o clasă fictivă denumită ExStatic astfel:

```
class ExStatic
{ public:
    static int n;
    ExStatic () { n++; } //constructor
    ~ExStatic () { n--; } //destructor
};
```

Clasa ExStatic are o dată membră statică (n), un constructor (care incrementează pe n la crearea unui obiect) și un destructor (care decrementează pe n la eliminarea unui obiect). Să utilizăm această clasă în diferite moduri:

```
int ExStatic::n=0;           //definire si initializare

void f()
{ ExStatic ob;             //se creeaza obiectul local ob
    cout << ob.n << ' ';
}

int main ()
{ExStatic a;               //creez un obiect
    cout << ExStatic::n << ' '; //scric 1
    ExStatic b[5];            //creez un vector cu 5 obiecte
    cout << ExStatic::n << ' '; //scric 6
    ExStatic* c=new ExStatic; //creez dinamic un obiect
    cout<<a.n<<' ';
    delete c;                //eliberez dinamic obiectul c
    cout << ExStatic::n << ' '; //scric 6
    f();                     //apelez functia f
    cout << ExStatic::n << ' '; //scric 6
    return 0;
}
```

Executând programul, se va afișa: 1 6 7 6 7 6

Observații

1. La membrul static n ne-am referit în două moduri diferite (ExStatic::n și prin intermediu unui obiect a.n sau ob.n). Indiferent de modul de referire, valoarea la care ne referim este aceeași.
2. Definirea și inițializarea membrului static n s-a realizat în afara clasei.
3. Datele statice există, chiar dacă nu există obiecte ale clasei respective.

Funcțiile membre statice

Funcțiile membre statice efectuează operații care nu sunt asociate obiectelor individuale, ci întregii clase. Funcțiile membre statice se referă numai la date statice și nu se pot referi la `this` (deoarece `this` este definit doar în cadrul obiectelor, reprezentând adresa obiectului curent).

Exemplu

Pentru exemplificare, vom adapta clasa fictivă precedentă `ExStatic`. Mai exact, vom transforma data membru statică din membru public în membru privat și vom adăuga funcții publice de acces (o funcție statică și o funcție nestatică). Vom adăuga și o dată membru nestatică, pentru a evidenția posibilele erori.

```
class ExStatic
{
private:
    static int n;
    int m;
public:
    ExStatic () { n++; } //constructor
    ~ExStatic () { n--; } //destructor
    void Set (int a, int b) { n=a; m=b; }
    static void Scrie() { cout<<n<<' ';}
};
```

Pentru a afișa valoarea membrului static privat `n`, se va utiliza funcția statică `Scrie()`. Această funcție accesează doar membrul static `n` (fiind o funcție statică). Funcția `Set` este nestatică, deci poate fi accesată atât membrii statici cât și membrii nestatici ai clasei.

```
int ExStatic::n=0;           //definirea datei membre statice

void f()
{ ExStatic ob;
  ob.Scrie();               //scrie 7
}

int main ()
{ExStatic a;                  //creez un obiect
  ExStatic::Scrie();          //scrie 1
  ExStatic b[5];              //creez un vector cu 5 obiecte
  ExStatic::Scrie();          //scrie 6
  ExStatic* c=new ExStatic;   //creez dinamic un obiect
  a.Scrie();                  //scrie 7
  delete c;
  ExStatic::Scrie();          //scrie 6
  f();
  ExStatic::Scrie();          //scrie 6
  a.Set(100, 200);
  ExStatic::Scrie();          //scrie 100
  return 0;
}
```

Executând programul, se va afișa: 1 6 7 6 7 6 100

Observații

1. La funcția statică `Scrie()` ne-am referit în două moduri diferite (`ExStatic::Scrie()` și prin intermediul unui obiect `a.Scrie()` sau `ob.Scrie()`). O funcție membră nestatică se utilizează doar prin intermediul unui obiect.
2. Dacă în funcția `Scrie()` am fi încercat să afișăm și valoarea membrului nestatic `m`, am fi obținut o eroare la compilare.

2.14. Specifierul `const`

Specifierul `const` poate fi utilizat în diferite contexte, dar efectul este similar în orice context: elementul asupra căruia acționează `const` nu mai poate fi modificat ulterior.

Specifierul `const` aplicat variabilelor

Specifierul `const` poate fi utilizat la *declararea variabilelor* astfel:

```
| const tip var=valoare;
```

Drept urmare, valoarea variabilei `var` nu se poate modifica ulterior.

Specifierul `const` aplicat parametrilor funcțiilor

Specifierul `const` poate precedea declararea parametrilor funcțiilor transmiși prin adresă (pointeri sau referințe) sub forma:

```
| tip numefunctie(const tip& numeparametru, ...);
```

Drept urmare, în interiorul funcției valoarea parametrului respectiv nu poate fi modificată. Utilizând specifiatorul `const`, compilatorul verifică faptul că valorile care nu trebuie să fie modificate nu vor fi modificate, iar programatorul este asigurat că, prin apelarea funcției, valoarea parametrului respectiv nu se modifică. Prin urmare, ca regulă generală, întotdeauna este bine să utilizați `const` pentru un parametru de tip referință a cărui valoare nu doriți să fie modificată.

Specifierul `const` aplicat obiectelor

Specifierul `const` poate fi utilizat și la declararea unui obiect, sub forma:

```
| const NumeClasă NumeObiect;
```

Ca urmare, toți membrii date ai obiectului nu mai pot fi modificați. Din acest motiv, este obligatoriu ca la declararea obiectului să fie utilizat un constructor care să inițializeze toate datele membre.

Obiectele declarate const nu pot fi transmise prin referințe care nu sunt de asemenea const.

Exemplu

Pentru exemplificare, vom crea o clasă fictivă, denumită ExConst, care are o singură dată membră (x), un constructor care inițializează pe x, precum și funcții de acces la x :

```
class ExConst
public:
    int x;
    ExConst(int a=0) { x = a; }           //constructor
    void ResetValue() { x = 0; }
    void SetValue(int y) { x = y; }
    int GetValue() const { return x; }
};
```

Să considerăm obiectul ob declarat astfel :

```
const ExConst ob(7);
```

Următoarele trei instrucțiuni reprezintă tentative ilegale de a modifica valoarea lui x, prin urmare, vor genera erori la compilare :

```
ob.x = 5;
ob.ResetValue();
ob.SetValue(5);
```

Specificatorul const aplicat funcțiilor membre ale claselor

Specificatorul const poate fi aplicat funcțiilor membre ale claselor, menționându-l imediat după antetul funcției, atât în declarația funcției :

```
tip NumeFuncție(listă_parametri) const;
```

cât și în definiția funcției :

```
tip NumeClasă::NumeFuncție(listă_parametri) const
{ ... }
```

Specificatorul const asigură compilatorul că funcția nu va modifica datele membre ale clasei. Cu obiectele declarate const se pot utiliza doar funcții declarate const.

Constructorii nu trebuie declarați const.

ri diferite
rie() sau
intermediul

ilui nestatic

este similar
iai poate fi

lor transmiși

v nu poate fi
ul că valorile
te asigurat că,
Prin urmare,
parametru de

b formă :

2.15. Funcțiile friend

O funcție friend poate accesa membrii private și protected ai clasei pentru care funcția este declarată friend. Pot fi declarate friend atât funcții obișnuite, cât și funcții membre ale altor clase.

Pentru a declara faptul că o funcție este funcție prietenă (friend) a unei clase, trebuie să includem în interiorul clasei declarația funcției, precedată de cuvântul-cheie friend. Funcțiile friend nefiind funcții membre ale clasei, specificatorii de acces nu au niciun efect.

Cuvântul-cheie friend se specifică doar la declarația funcției în interiorul clasei, nu și la definirea acesteia.

O aceeași funcție poate fi funcție friend pentru mai multe clase și aceeași clasă poate avea mai multe funcții friend.

Exemplu

Pentru exemplificare, vom declara o clasă fictivă denumită ExFFriend. În această clasă vom declara o dată membră privată x și două funcții de acces publice (Setx() și Getx()). Funcția Inc este declarată funcție prietenă a clasei ExFFriend.

```
class ExFFriend
{
private:
    int x;
public:
    int Getx() { return x; }
    void Setx(int a) { x=a; }
    friend void Inc(ExFFriend &);
};
```

Funcția Inc va incrementa valoarea datei membru x. Acest lucru este posibil deoarece funcția fiind declarată friend, are acces și la membrii privați ai clasei.

```
void Inc(ExFFriend & ob)
{ ob.x++; }
```

Observați că funcțiile prietene, nefiind funcții membre, trebuie să primească obiectul ca parametru. Iată cum putem utiliza funcția Inc :

```
ExFFriend a;
a.Setx(7);
Inc(a);
cout<<a.Getx();
```

Evident, se va afișa pe ecran valoarea 8.

2.16. Clasele friend

În mod similar cu funcțiile friend, putem declara clase friend. Pentru a declara faptul că o clasă A este prietenă (friend) cu o clasă B, vom include în clasa B o declarație de forma :

```
friend class A;
```

Ca urmare, clasa A va avea acces la membrii private și protected ai clasei B. Cu alte cuvinte, toate funcțiile membre ale clasei A sunt prietene cu clasa B.

Exemplu

Vom declara o clasă A și o clasă B. Clasa A va fi declarată clasă prietenă a clasei B. În clasa A se află o funcție membră publică, denumită Suma () care va accesa o dată membru privată a clasei B.

```
class B;

class A
{
private:
    int y;
public:
    A(int b=0) { y=b; }
    void Suma(B & ob);
};

class B
{
private:
    int x;
public:
    int Getx() { return x; }
    void Setx(int a) { x=a; }
    friend class A;
};
void A::Suma (B & ob)
{ ob.x+=y; }
```

Observați că am utilizat o declarație *forward* a clasei B înainte de a declara clasa A. Această declarație anunță compilatorul că există clasa B și va fi definită ulterior.

Iată și un exemplu de utilizare a celor două clase :

```
A a(10);
B b;
b.Setx(7);
a.Sum(a);
cout<<b.Getx();
```

Evident, această secvență de instrucțiuni va afișa pe ecran valoarea 17.

2.17. Supraîncărcarea operatorilor

Supraîncărcarea operatorilor este un concept similar cu supraîncărcarea funcțiilor. Pentru a supraîncărca un operator, vom defini o funcție cu numele operator simbol, unde operator este un cuvânt-cheie, iar simbol este simbolul asociat operatorului.

Pot fi supraîncărcați doar operatorii existenți în limbajul C++ și doar pentru a fi utilizati cu clase. Aproape toți operatorii limbajului pot fi supraîncărcați, excepțiile fiind . (operatorul de selecție directă), * (operatorul unar de dereferențiere), :: (operatorul de rezoluție), ?: (operatorii condiționali) și operatorul sizeof. Prin supraîncărcare, nici prioritatea, nici asociativitatea operatorilor nu se schimbă.

Dacă operatorul de atribuire = nu este explicit supraîncărcat, compilatorul generează automat un operator de atribuire care va copia în obiectul din stânga operatorului = valorile datelor membre din obiectul aflat în dreapta operatorului =.

Operatorii pot fi supraîncărcați fie ca funcții membre ale clasei, fie ca funcții friend. Un operator unar poate fi supraîncărcat ca funcție membră fără niciun parametru (el acționând asupra obiectului curent) sau ca funcție friend cu un parametru. Un operator binar poate fi supraîncărcat ca funcție membră cu un singur parametru (primul operand fiind considerat obiectul curent, iar al doilea operand este parametrul) sau ca funcție friend cu doi parametri.

Operatorii de atribuire (= += -= *= /= %= ^= &= |= <<= >>=), operatorul de selecție indirectă -> și operatorul de indexare [] pot fi supraîncărcați doar ca funcții membre ale clasei.

Operatorul de citire >> și operatorul de scriere << pot fi supraîncărcați doar ca funcții prietene.

Ceilalți operatori pot fi supraîncărcați fie ca funcții membre, fie ca funcții prietene, dar în general se respectă următoarele reguli :

- operatorii unari sunt supraîncărcați ca funcții membre ;
- operatorii binari care nu modifică niciunul dintre operanzi sunt supraîncărcați ca funcții prietene ;
- un operator binar care modifică unul dintre operanzi e preferabil să fie funcție membră.

Observații

1. Când supraîncarcăm un operator este bine ca prin aceasta să menținem semnificația sa inițială. În caz contrar, ar fi mai bine să definim o funcție membră cu un nume sugestiv care să realizeze prelucrările dorite.
2. Este bine să venim în întâmpinarea așteptărilor utilizatorilor. Mai exact, să supraîncarcăm toți operatorii similari : dacă am supraîncărcat operatorii aritmetici sau pe biți, supraîncarcăm și operatorii de atribuire compuși corespunzători (de exemplu, dacă am supraîncărcat operatorul +, supraîncarcăm și +=); dacă supraîncarcăm operatorii relaționali și de egalitate, îi supraîncarcăm pe toți (și == și !=; și < și > și >= și <=); dacă supraîncarcăm ++ și -- în formă prefixată, atunci îi supraîncarcăm și în formă postfixată.

Utilizarea operatorilor supraîncărcați

Operatorii supraîncărcați pot fi utilizați în două moduri (uzual, ca un operator sau ca apel de funcție). Pentru a explica, vom nota cu α un operator unar, cu β un operator binar, iar cu ob și $ob1$ două obiecte. Uzual, operatorul unar α poate fi utilizat în formă prefixată αob . Doar operatorii $++$ și $--$ supraîncărcați pot fi utilizati și în formă postfixată ($ob++$ sau $ob--$). Operatorul binar β poate fi utilizat uzual $ob \beta ob1$.

Operatorii supraîncărcați pot fi utilizati și printr-un apel de funcție. Dacă operatorii sunt funcții membre ale clasei, apelul se va realiza astfel: $ob.\text{operator } \alpha()$, respectiv $ob.\text{operator } \beta(ob1)$. Dacă operatorii sunt funcții prietene ale clasei, atunci apelul funcției se realizează astfel: $\text{operator } \alpha(ob)$ sau $\text{operator } \beta(ob, ob2)$.

Exemplul 1. Supraîncarcarea operatorilor ca funcții membre pentru clasa mesaj

Pentru a exemplifica supraîncarcarea operatorilor ca funcții membre vom relua clasa *mesaj*, prezentată la începutul acestui capitol, și vom supraîncărca următorii operatori pentru aceasta :

- operatorul unar \sim (care va inversa mesajul)
- operatorul de atribuire $=$
- operatorul $+$ (care va avea rolul de a concatena două mesaje, obținând rezultatul intr-un alt mesaj)
- operatorul $+=$ (care va concatena două mesaje, obținând rezultatul în primul dintre acestea)
- operatorul $==$ (care verifică dacă două mesaje sunt egale)
- operatorul $!=$ (care verifică dacă două mesaje sunt diferite)
- operatorul de indexare $[]$

Declarația clasei mesaj (în fișierul mesaj.h)

```
class mesaj
{
private:
    char * sir;
    int lg;
public:
    mesaj();           //constructor fara parametri
    mesaj (char *);  //constructor supraincarcat
    ~mesaj();         //destructor
    mesaj(mesaj&);   //constructor de copiere

    int GetLg();
    void Scrie();
}
```

```

void Majuscule();
//operatori supraincarcati
mesaj operator~();
mesaj& operator=(const mesaj &);
const mesaj operator+(const mesaj&) const;
const mesaj& operator+=(const mesaj&);
bool operator==(const mesaj&) const;
inline bool operator!=(const mesaj&) const;
char operator[] (const int) const;
};

```

Implementarea operatorilor ca funcții membre (în fișierul mesaj.cpp)

- Definirea operatorului unar ~ :

```

mesaj mesaj::operator~()
{
    int st, dr;
    char aux;
    for (st=0, dr=lg-1; st<dr; st++, dr--)
        {aux=sir[st]; sir[st]=sir[dr]; sir[dr]=aux; }
    return *this;
}

```

Operatorul ~ poate fi utilizat doar în formă prefixată. De exemplu :

```

mesaj m("12345ABC");
~m;
m.Scrie();

```

Executând această secvență de instrucțiuni, pe ecran se va afișa :

CBA54321

Ca apel de funcție, operatorul ~ ar putea fi utilizat (cu același efect) astfel :

```
m.operator~();
```

- Definirea operatorului de atribuire = :

```

mesaj& mesaj::operator=(const mesaj & m)
{
    //verificam daca este autoatribuire
    if (this == &m) return *this;
    //eliboram eventuala zona de memorie alocata deja
    if (sir) delete sir;
    //alocam o noua zona de memorie pentru obiectul curent
    sir = new char[m.lg];
    //copiem in obiectul curent obiectul m
    lg=m.lg;
    for (int i=0; i<lg; i++) sir[i]=m.sir[i];
    return *this;
}

```

Operatorul de atribuire poate fi supraîncărcat doar ca funcție membră și prototipul său va fi întotdeauna :

```
| NumeClasă& NumeClasă::operator=(const NumeClasă &);
```

Observați că tipul rezultatului este o referință la un obiect al clasei. Acest lucru este necesar pentru a putea utiliza operatorul de atribuire în mod înlănituit, sub forma: `ob1=ob2=...=obk`. Din acest motiv, funcția trebuie să returneze la final obiectul curent (`return *this;`).

Deoarece pentru clasa mesaj constructorii alocă dinamic memorie, iar destrutorul eliberează dinamic memoria alocată, mai sunt două aspecte la care trebuie să acordăm atenție când implementăm operatorul de atribuire.

Un prim aspect constă în alocarea memoriei pentru rezultat. Rezultatul trebuie să fie stocat în obiectul curent, care este posibil să aibă deja memorie alocată. De aceea, trebuie mai întâi să eliberăm memoria deja alocată, să alocăm o altă zonă de memorie, suficientă pentru a reține rezultatul atribuirii, apoi să realizăm copierea.

Cel de-al doilea aspect se referă la atribuirile de forma `ob=ob`. Aceasta este o atribuire legală și operatorul nostru ar trebui să funcționeze. Dar acest caz trebuie tratat separat (`if (this == &m) return *this;`); în caz contrar, vom elibera memoria alocată obiectului curent (care de data aceasta coincide cu obiectul care se copiază și, drept urmare, obiectul de copiat va fi distrus).

- Definirea operatorului de concatenare +:

```
| const mesaj mesaj::operator+(const mesaj & m) const
{mesaj temp;
 int i;
 temp.lg=lg+m.lg;
 temp.sir=new char [temp.lg];
 for (i=0; i<lg; i++) temp.sir[i]=sir[i];
 for (i=0; i<m.lg; i++) temp.sir[i+lg]=m.sir[i];
 return temp;
}
```

Observați că am creat un obiect temporar `temp` în care construim rezultatul concatenării; mai întâi alocăm dinamic memoria necesară, apoi copiem caracterele din mesajul curent, apoi în continuare caracterele din mesajul transmis ca parametru. La final returnăm ca rezultat obiectul temporar.

Explicații suplimentare sunt necesare referitor la utilizarea specificatorului `const`. Aplicat parametrului (`const mesaj & m`), indică faptul că operatorul nu modifică parametrul `m`, transmis prin referință. Aplicat operatorului (`const` după antet) indică faptul că funcția nu modifică datele membre ale obiectului curent. Aplicat rezultatului funcției (`const mesaj`), împiedică scrierea unor construcții ciudate de tipul: `m1+m2=m3`.

Operatorul + poate fi utilizat, de exemplu, astfel:

```
| mesaj m0, m1("Ana "), m2("are mere");
m0=m1+m2;
m0.Scrie();
```

După execuția acestor instrucțiuni, pe ecran se va afișa : Ana are mere

Ca apel de funcție, operatorul poate fi utilizat (cu același efect) și astfel :

`m0=m1.operator +(m2);`

Operatorul + poate fi utilizat și astfel :

```
mesaj m0, m1("Ana ");
m0=m1+"are mere";
m0.Scrie();
```

La o astfel de utilizare, se realizează o **conversie implicită**. Sirul de caractere "are mere" este convertit într-un obiect temporar, prin apelarea automată a construcțorului corespunzător. Dar – atenție ! – această conversie este posibilă deoarece există un constructor care creează un obiect de tip mesaj pe baza unui sir de caractere. Practic, construcția `m1+"are mere"` este echivalentă cu `m1+mesaj ("are mere")`.

Însă o utilizare de forma :

```
m0=" are mere"+m1;
```

este ilegală și va genera o eroare la compilare. Implementând operatorul + ca funcție membră, el nu mai este comutativ, primul său operand fiind obligatoriu un obiect. Din acest motiv, este recomandabil ca operatorul + să fie implementat ca funcție prietenă.

- Definirea operatorului de atribuire compus `+=`:

```
const mesaj& mesaj::operator+=(const mesaj& m)
{mesaj temp;
 int i;
 temp.lg=lg+m.lg;
 temp.sir=new char [temp.lg];
 for (i=0; i<lg; i++) temp.sir[i]=sir[i];
 for (i=0; i<m.lg; i++) temp.sir[i+lg]=m.sir[i];
 *this=temp;
 return *this;
}
```

Diferența dintre operatorul + și operatorul += constă în faptul că rezultatul concatenării trebuie să fie plasat în obiectul curent. Din nou, pentru a putea utiliza acest operator înlănțuit, am returnat o referință la rezultat (`return *this;`). Funcția modifică obiectul curent, din acest motiv nu apare specificatorul `const` după antetul funcției.

Observație

În cazul operatorului +, tipul rezultatului este un obiect. Returnarea unei copii a obiectului rezultat este obligatorie, deoarece rezultatul a fost obținut într-o variabilă locală, care s-ar fi distrus la ieșirea din funcție. În cazul operatorului +=, tipul rezultatului este o referință la un obiect. Nu a fost necesară copierea obiectului, deoarece rezultatul se află în obiectul curent, care nu este distrus la terminarea execuției funcției, deci a fost suficientă transmiterea ca rezultat a unei referințe la obiectul curent (mai eficient).

nere
fel:
de caractere
îtă a construc-
eoarece există
ctere. Practic,
ere").

- Un exemplu de utilizare a operatorului `+=`:

```
mesaj m0("ana"), m1(" are mere"), m2(" bune");
m0+=m1+=m2;
m0.Scrie();
m1.Scrie();
```

Asociativitatea operatorilor de atribuire este de la dreapta la stânga. Prin urmare, mai întâi se concatenează `m2` la `m1`, apoi noul `m1` la `m0`. După executarea atribuirilor, pe ecran se va afișa :

ana are mere bune
are mere bune

- Definirea operatorului de egalitate `==`:

```
bool mesaj::operator ==(const mesaj& m) const
{
    int i;
    if (lg!=m.lg) return false;
    for (i=0; i<lg && sir[i]==m.sir[i]; i++);
    return (i==lg);
}
```

Pentru ca două mesaje să fie considerate egale, ele trebuie să conțină exact aceleasi caractere.

- Definirea operatorului de inegalitate `!=`:

```
inline bool mesaj::operator !=(const mesaj& m) const
{
    return !(*this == m);
```

Observați că pentru a implementa operatorul `!=` am utilizat operatorul `==`. Pentru eficiență, am declarat operatorul `!=` drept funcție inline.

- Definirea operatorului de indexare `[]`:

```
char mesaj::operator[] (const int poz) const
{
    return sir[poz];
}
```

Odată supraîncărcat operatorul de indexare, putem accesa orice caracter din vectorul de caractere din mesaj :

```
mesaj m0("anapoda");
cout<<m0[3];
```

Pe ecran se va afișa litera p. Dar orice încercare de a modifica un caracter din vector va genera eroare la compilare.

că rezultatul
putea utiliza
n `*this;`).
const după

unei copii a
tr-o variabilă
lui `+=`, tipul
a obiectului,
a terminarea
i referințe la

Exemplul 2. Supraîncărcarea operatorilor pentru clasa fractie

Pentru a exemplifica supraîncărcarea operatorilor ca funcții friend vom relua clasa `fractie`, adăugând declarații friend pentru următorii operatori:

- operatorul – unar (schimbă semnul algebric al fracției)
- operatori aritmetici (+, -, *, /)
- operatori de atribuire compuși cu operatorii aritmetici (+=, -=, *=, /=)
- operatori de egalitate (==, !=)
- operatori relationali (<, >, <=, >=)
- operatorul de citire >>
- operatorul de afișare <<

De asemenea, vom supraîncărca drept funcții membre ale clasei operatorii de incrementare/decrementare, în formă prefixată și în formă postfixată (++ , --), precum și operatorul de conversie explicită din fracție în număr real de tip `double`.

Nu vom supraîncărca operatorul de atribuire =, întrucât acesta va fi automat supraîncărcat (realizându-se copierea membru cu membru, ceea ce este suficient în acest caz).

Declararea clasei fractie

```
class fractie
{
private:
    int a;                                //numărator
    int b;                                //numitor
public:
    int GetNumarator(){ return a; }
    int GetNumitor() { return b; }
    void SetNumitor(int);
    void SetNumarator(int);
    void Simplifica();
    fractie(int x=0, int y=1); //constructorul clasei

    //operatori aritmetici
    friend fractie operator-(const fractie &);
    friend fractie operator+(const fractie &, const fractie &);
    friend fractie operator-(const fractie &, const fractie &);
    friend fractie operator*(const fractie &, const fractie &);
    friend fractie operator/(const fractie &, const fractie &);

    //operatorii de atribuire compusi
    friend fractie& operator+=(fractie &, const fractie &);
    friend fractie& operator-=(fractie &, const fractie &);
    friend fractie& operator*=(fractie &, const fractie &);
    friend fractie& operator/=(fractie &, const fractie &);

    //operatorii de egalitate
    friend bool operator==(const fractie &, const fractie &);
    friend bool operator!=(const fractie &, const fractie &);

    //operatorii relationali
```

```

friend bool operator<(const fractie &, const fractie &);
friend bool operator>(const fractie &, const fractie &);
friend bool operator<=(const fractie &, const fractie &);
friend bool operator>=(const fractie &, const fractie &);
    //operatorii de citire si scriere
friend istream & operator>>(istream &, fractie &);
friend ostream & operator<<(ostream &, const fractie &);

    //operatori supraincarcati ca functii membre
fractie& operator++(); //forma prefixata
fractie operator++(int fictiv); // forma postfixata
fractie& operator--(); //forma prefixata
fractie operator--(int fictiv); // forma postfixata

operator double() const; //conversie explicita la double
};

```

Definițiile operatorilor suprâncărcați

Operatorul – unar returnează o fracție cu semnul algebric schimbat.

Atenție! Obiectul transmis ca parametru nu trebuie să se modifice! Prin urmare utilizăm un obiect de tip fractie temporar în care construim fracția cu semn schimbat și returnăm acest obiect ca rezultat. Rezultatul funcției este de tip fractie și nu fractie & (referință), deoarece este necesară copierea obiectului temporar care conține rezultatul (la terminarea execuției funcției obiectul temporar este distrus și transmiterea unei referințe către acesta nu ar avea sens).

```

fractie operator-(const fractie & f)
{
    fractie temp(-f.a, f.b);
    return temp;
}

```

Operatorii aritmetici implementează operațiile cu fracții binecunoscute de la matematică :

$$\frac{a_1}{b_1} + \frac{a_2}{b_2} = \frac{a_1 \cdot b_2 + b_1 \cdot a_2}{b_1 \cdot b_2}$$

$$\frac{a_1}{b_1} - \frac{a_2}{b_2} = \frac{a_1 \cdot b_2 - b_1 \cdot a_2}{b_1 \cdot b_2}$$

$$\frac{a_1}{b_1} \cdot \frac{a_2}{b_2} = \frac{a_1 \cdot a_2}{b_1 \cdot b_2}$$

$$\frac{a_1}{b_1} : \frac{a_2}{b_2} = \frac{a_1 \cdot b_2}{b_1 \cdot a_2}$$

Toți cei patru operatori primesc drept parametri două obiecte de tip fractie (care nu se modifică, de aceea este utilizat specificatorul const), calculează într-o variabilă temporară rezultatul operației, simplifică fracția rezultată și returnează rez-

ca rezultat al funcției. Justificarea faptului că operatorii aritmetici returnează un obiect ca rezultat, nu o referință la acesta, este aceeași ca în cazul operatorului unar -.

```

fractie operator+(const fractie &f1, const fractie &f2)
{
    fractie rez;
    rez.a=f1.a*f2.b+f1.b*f2.a;
    rez.b=f1.b*f2.b;
    rez.Simplifica();
    return rez;
}

fractie operator-(const fractie & f1, const fractie & f2)
{
    fractie rez;
    rez.a=f1.a*f2.b-f1.b*f2.a;
    rez.b=f1.b*f2.b;
    rez.Simplifica();
    return rez;
}

fractie operator*(const fractie & f1, const fractie & f2)
{
    fractie rez;
    rez.a=f1.a*f2.a;
    rez.b=f1.b*f2.b;
    rez.Simplifica();
    return rez;
}

fractie operator/(const fractie & f1, const fractie & f2)
{
    fractie rez;
    rez.a=f1.a*f2.b;
    rez.b=f1.b*f2.a;
    rez.Simplifica();
    return rez;
}

```

Operatorii de atribuire compuși cu operatori aritmetici sunt similari operatorilor aritmetici, numai că în acest caz rezultatul se obține în primul parametru (acesta nu mai are specificatorul const, deoarece se modifică). Rezultatul returnat este o referință la primul parametru, pentru a putea utiliza operatorii de atribuire înlănțuit.

```

fractie& operator+=(fractie &f1, const fractie &f2)
{
    f1.a=f1.a*f2.b+f1.b*f2.a;
    f1.b=f1.b*f2.b;
    f1.Simplifica();
    return f1;
}

fractie& operator-=(fractie &f1, const fractie &f2)
{
    f1.a=f1.a*f2.b-f1.b*f2.a;
    f1.b=f1.b*f2.b;
}

```

ază un obiect
unar -.
&f2)

e & f2)

e & f2)

e & f2)

i operatorilor
ru (acesta nu
turnat este o
re înlăncuit.

```
f1.Simplifica();
return f1;
}

fractie& operator*=(fractie & f1, const fractie & f2)
{f1.a=f1.a*f2.a;
f1.b=f1.b*f2.b;
f1.Simplifica();
return f1;
}

fractie& operator/=(fractie & f1, const fractie & f2)
{f1.a=f1.a*f2.b;
f1.b=f1.b*f2.a;
f1.Simplifica();
return f1;
}
```

Operatorii de egalitate și operatorii relaționali primesc ca parametri referințe la două obiecte de tip `fractie` (care nu se modifică, din acest motiv am utilizat specificatorul `const`) și returnează un rezultat logic (true dacă relația este îndeplinită și false în caz contrar). Observați că în implementarea operatorilor `>` și `>=` am utilizat operatorul deja definit `<`, iar pentru a implementa operatorul `<=` am utilizat operatorul deja definit `>`.

```
bool operator==(const fractie & f1, const fractie & f2)
{ return f1.a*f2.b-f1.b*f2.a==0; }

bool operator!=(const fractie & f1, const fractie & f2)
{ return f1.a*f2.b-f1.b*f2.a; }

bool operator<(const fractie & f1, const fractie & f2)
{ return f1.a*f2.b<f1.b*f2.a; }

bool operator>(const fractie & f1, const fractie & f2)
{ return f2<f1; }

bool operator<=(const fractie & f1, const fractie & f2)
{ return !(f1>f2); }

bool operator>=(const fractie & f1, const fractie & f2)
{ return !(f1<f2); }
```

Operatorul de citire `>>` poate fi supraîncărcat pentru a citi numărătorul și numitorul unei fracții. Acest operator are întotdeauna doi parametri: o referință la un obiect de tip `istream` (fluxul de intrare) și o referință la obiectul care va fi citit. Ambii parametri se modifică în urma citirii. Rezultatul este de tip `istream&` (referință la fluxul de intrare), pentru a putea utiliza înlăncuit acest operator.

```
istream & operator>>(istream & in, fractie & f)
{
    in>>f.a>>f.b;
    return in;
}
```

Operatorul de scriere `<<` va fi supraîncărcat în mod similar, pentru a afișa fracția într-un mod convenabil. Acest operator are întotdeauna doi parametri: referință către fluxul de ieșire (`ostream&`) și referință la obiectul care se afișează (acesta nu se modifică). Rezultatul este de tip `ostream&` (se returnează fluxul de ieșire), pentru a putea utiliza înlățuit acest operator.

```
ostream & operator<<(ostream &out, const fractie &f)
{
    out<<f.a<<'/'<<f.b;
    return out;
}
```

Operatorul de incrementare în formă prefixată va mări cu 1 valoarea fracției din obiectul curent și va returna o referință la obiectul curent, care reține fracția incrementată:

```
fractie& fractie::operator++() //forma prefixata
{
    a+=b;
    return *this;
}
```

În mod similar se implementează și operatorul de decrementare `--` în formă prefixată:

```
fractie& fractie::operator--() //forma prefixata
{
    a-=b;
    return *this;
}
```

Pentru a exemplifica modul de funcționare a operatorului `++` în formă prefixată, să analizăm următorul exemplu:

```
fractie f0, f1(3,4);
f0=++f1;
cout<<f0<<" "<<f1;
```

Se mărește mai întâi cu 1 valoarea fracției memorate în obiectul `f1` ($3/4$), apoi această valoare se atribuie variabilei `f0`. Ca urmare, pe ecran se va afișa: $7/4$ $7/4$

Operatorul de incrementare în formă postfixată trebuie să adune 1 la fracția din obiectul curent, dar trebuie să returneze ca rezultat fracția dinainte de modificare. Prin urmare, este necesară o variabilă locală `temp` în care se reține fracția curentă înainte de modificare, se incrementează fracția curentă, apoi se returnează ca rezultat fracția memorată în `temp`. Pentru a face distincția între forma prefixată și forma postfixată, în formă postfixată se transmite un parametru suplimentar de tip `int` (fictiv), care nu este utilizat.

```
fractie fractie::operator++(int fictiv) // forma postfixata
{
    fractie temp(a,b);
    ++(*this);
    return temp;
}
```

Să exemplificăm modul de funcționare a acestui operator :

```
fractie f0, f1(3,4);
f0=f1++;
cout<<f0<<" "<<f1;
```

Pe ecran se va afișa $3/4 \quad 7/4$ deoarece f_1 este incrementat, dar lui f_0 îi se atribuie valoarea dinaintea incrementării.

În mod similar se implementează operatorul de decrementare în formă postfixată :

```
fractie fractie::operator--(int fictiv) // forma postfixata
{
    fractie temp(a,b);
    --(*this);
    return temp;
}
```

Observați că la implementare am utilizat operatorul $++$, respectiv $--$ în formă prefixată, deja definit.

Analizând comparativ cele două implementări (pentru forma prefixată și pentru cea postfixată), deducem că forma prefixată este mai eficientă. Prin urmare, atunci când putem utiliza oricare dintre cele două forme, este convenabil să alegem forma prefixată. Această observație este general valabilă.

Am văzut în exemplul precedent (clasa mesaj) că se realizează **conversii implicite** de la un tip predefinit al limbajului către un obiect, utilizându-se constructorii definiți ai clasei. Pentru a realiza însă conversii de la un obiect al clasei către un tip predefinit al limbajului trebuie să supraîncărcăm operatorii de **conversie explicită** (*cast*). Operatorii de conversie explicită au o serie de particularități :

- au ca nume un tip ;
- nu au tip rezultat (acesta fiind implicit același cu numele operatorului) ;
- nu au parametri ;
- trebuie să fie funcții membre ale clasei.

Vom implementa ca funcție membră un operator de conversie explicită, care transformă un obiect de tip fractie într-un număr real de tip double :

```
fractie::operator double() const
{ return (double) a/b; }
```

Un exemplu de utilizare a acestui operator :

```
fractie f1(3,4);
cout<<f1<<" =" <<(double)f1;
```

Pe ecran se va afișa : $3/4=0.75$

2.18. Tratarea erorilor

La școală învățăm că „datele de intrare sunt corecte”, prin urmare, nu este necesar să validăm că datele de intrare respectă restricțiile problemei. Din păcate, în realitate nu e deloc aşa. De exemplu, dacă lucrăm la o aplicație în care utilizatorul trebuie să introducă numărul de telefon, chiar dacă îi explicăți printr-un mesaj cum trebuie să introducă numărul de telefon, și oferiți chiar și un exemplu, este extrem de probabil să existe utilizatori care să introducă numărul de telefon în cele mai variate moduri: 0740-123-456 sau 0740.123.456 sau 0740 123 456 sau oricum altcumva. În viața reală trebuie să validăm datele și să ne asigurăm că sunt respectate restricțiile necesare unei bune funcționări a aplicației. Dacă dorim să citim dintr-un fișier, trebuie să ne asigurăm că fișierul există; dacă dorim să citim un număr pozitiv, trebuie să ne asigurăm că utilizatorul a introdus un număr și că numărul este pozitiv etc.

Astfel de erori, care pot apărea în timpul execuției unui program, sunt denumite de programatori **excepții** (deoarece acestea sunt cazuri particulare, ce nu ar trebui să apară în mod normal).

O variantă posibilă de tratare a erorilor ar arăta astfel:

```
secventa de instructiuni 1;
daca a aparut o eroare atunci trateaza eroarea 1
    altfel
        secventa de instructiuni 2;
        daca a aparut o eroare, trateaza eroarea 2
        ...
    
```

O astfel de abordare este posibilă, dar nerecomandabilă. Codul devine extrem de stufoș, greu de înțeles, greu de întreținut, deoarece codul de executat este întrețesut cu codul de tratare a erorilor.

Experiența programatorilor *POO* recomandă o abordare diferită în tratarea erorilor: **codul de executat trebuie să fie separat de codul de tratare a erorilor**.

Când se identifică o eroare se „aruncă” (*throw*) o valoare corespunzătoare erorii respective. „Aruncarea” unei valori pentru o excepție se realizează astfel:

```
throw(valoare_excepție);
```

Valoarea „aruncată” pentru o excepție poate fi de orice tip (număr, sir de caractere). De exemplu, să scriem o funcție care citește numărul de telefon al unui utilizator:

```
int NrTelefon()
{
    int lg, i;
    char s[100];
    cout<<"Introduceti numarul de telefon! "; cin>>s;
    lg=strlen(s);
    if (lg!=10)
        throw("Nr invalid: trebuie sa contine 10 cifre!");
    if (s[0]!='0')
        throw("Nr invalid: prima cifra trebuie sa fie 0");
```

```

for (i=0; i<lg; i++)
    if (! (s[i]>='0' && s[i]<='9'))
        throw("Nr invalid: trebuie sa contine doar cifre");
    return atoi(s);
}

```

Observați că la citirea numărului de telefon identificăm erorile posibile și „aruncăm” o excepție pentru fiecare dintre ele. Valoarea aruncată este de fiecare dată un sir de caractere, conținând un mesaj specific. Este posibil ca valorile „aruncate” să fie de tipuri diferite.

Când se „aruncă” o excepție, execuția programului este automat întreruptă. Pentru a evita acest lucru, trebuie să „capturăm” erorile. În acest scop, atunci când o secvență de cod este susceptibilă să genereze erori, ea va fi „încercată”, adică va fi inclusă într-un bloc `try`, astfel :

```

try
{
    secvență de instrucțiuni de executat
}

```

Acoladele sunt obligatorii în blocul `try`, chiar dacă secvența constă dintr-o singură instrucțiune.

După blocul `try` putem captura erorile cu ajutorul unui blocurilor `catch`. După `try` trebuie să apară un bloc `catch` pentru fiecare tip de eroare „aruncată”. Sintaxa unui bloc `catch` este :

```

catch (tip valoare)
{
    secvența de instrucțiune care se execută dacă este
    capturată o eroare
}

```

De exemplu, iată o încercare de a citi un număr de telefon, utilizând funcția `NrTelefon()` precedentă :

```

int nrt;
try
{
    nrt=NrTelefon();
}
catch (const char * s)
{
    cout<<"Eroare la introducerea numarului de telefon: ";
    cout<<s<<'\n';
}

```

Dacă secvența de instrucțiuni de executat aruncă erori de tipuri diferite, atunci trebuie să existe o secțiune `catch` pentru fiecare tip de eroare. În acest caz, toate erorile aruncate au fost de același tip (sir de caractere), deci a fost necesară doar o singură secțiune `catch` pentru a captura eroarea.

Dacă există mai multe secțiuni catch, atunci ele sunt parcuse în ordine, până se identifică prima secțiune pentru care tipul parametrului corespunde cu tipul erorii „aruncate”.

De exemplu, vom face o funcție de citire a vârstei, care „aruncă” erori de tipuri diferite (un sir de caractere, dacă vârsta nu este numerică, respectiv un număr întreg reprezentând vârsta, dacă vârsta nu este în intervalul [18, 99]):

```
int Varsta()
{
    int v, i;
    char s[100];
    cout<<"Introduceti varsta! (0-99)"; cin>>s;
    for (i=0; s[i]; i++)
        if (! (s[i]>='0' && s[i]<='9'))
            throw("Varsta trebuie sa contine doar cifre");
    v=atoi(s);
    if (v<18 || v>99) throw(v);
    return v;
}
```

Putem „încerca” să citim vârsta astfel:

```
int v;
try
{
    v=Varsta();
}
catch (const char * s)
{
    cout<<"A aparut o eroare la introducerea varstei: ";
    cout<<s<<'\n';
}
catch (const int val)
{
    cout<<"Pentru permis varsta este intre 18 și 99 ani, nu ";
    cout<<val<<" ani\n";
}
```

Observați că s-a utilizat o secțiune catch cu parametrul sir de caractere și o secțiune catch cu parametrul număr întreg.

Există posibilitatea de a utiliza și o secțiune catch implicită, aceasta se plasează ultima și se va executa dacă niciuna dintre secțiunile catch precedente nu se „potrivește” cu eroarea „aruncată”. Secțiunea catch implicită nu are parametru între paranteze, ci puncte de suspensie: catch(...).

ordine, până se
cu tipul erorii

erori de tipuri
n număr întreg

:e");

..";

ani, nu ";

caractere și o

sta se plasează
cedente nu se
are parametru

2.19. Aplicație. Numere naturale mari

Datele membre

Vom proiecta o clasă denumită `BigInt` pentru lucrul cu numere naturale mari scrise în baza 10. Numărul maxim de cifre ale numărului va fi o constantă simbolică `LGMAX`, definită în fișierul antet `bigint.h`.

Vom reprezenta un număr mare ca un vector `cifre`, în care vom reține cifrele numărului începând cu unitățile (în acest mod, poziția cifrei în vector va coincide cu puterea corespunzătoare a bazei). Vom reține de asemenea și numărul de cifre în data membră `lg`.

Funcțiile membre

Constructori

Vom implementa :

- un constructor care construiește un număr mare pe baza unui număr mic (număr ce poate fi memorat într-o variabilă de tip `int`) ; acest constructor va primi un parametru de tip `int`, parametru care are valoarea implicită 0 ;
- un constructor care va construi un număr mare pe baza unui număr memorat într-un sir de caractere ;
- un constructor de copiere.

Constructorii vor inițializa vectorul de cifre cu 0, pentru a ușura implementarea operatorilor.

Destructor

Nu este necesar să implementăm un destructor, compilatorul va genera automat un destructor.

Operatori

Vom implementa ca funcții membre operatorul de atribuire, operatorul de indexare, operatorii de incrementare și decrementare (în formă prefixată și în formă postfixată) și operatorii de conversie dintr-un număr mare într-un număr mic (de tip `int` sau `long long int`).

Ca funcții prietene ale clasei vom implementa operatorii aritmetici, operatorii de atribuire compuși cu operatorii aritmetici, operatori relationali, operatori de egalitate, operatori de citire scriere.

Declarația clasei BigInt

```

#define LGMAX 1000
class BigInt
{
private:
    char cifre[LGMAX]; //vectorul de cifre
    int lg;             //lungimea numarului

public:
    //constructori
    BigInt (int n=0);
    //constructor care creeaza numarul mare dintr-un nr. int
    BigInt(const char * s);
    //constructor care creeaza numarul mare dintr-un sir
    BigInt(BigInt&);
    //constructorul de copiere
    int GetLg() { return lg; }

    friend bool e0(const BigInt &);

    //operatorul de atribuire
    BigInt& operator = (const BigInt &);

    //operator de indexare
    int operator [] (const int) const;
    BigInt& operator++();           //forma prefixata
    BigInt operator++(int fictiv); // forma postfixata
    BigInt& operator--();           //forma prefixata
    BigInt operator--(int fictiv); // forma postfixata

    friend BigInt operator +(const BigInt &, const BigInt &);
    friend BigInt& operator +=(BigInt &, const BigInt &);

    friend BigInt operator -(const BigInt &, const BigInt &);
    friend BigInt& operator -=(BigInt &, const BigInt &);

    friend BigInt operator *(const BigInt &, const BigInt &);
    friend BigInt& operator *=(BigInt &, const BigInt &);

    friend BigInt operator /(const BigInt &, const BigInt &);
    friend BigInt& operator /=(BigInt &, const BigInt &);

    friend BigInt operator %(const BigInt &, const BigInt &);
    friend BigInt& operator %=(BigInt &, const BigInt &);

    //operatori de egalitate
    friend bool operator ==(const BigInt &, const BigInt &);
}

```

```

friend bool operator != (const BigInt &, const BigInt &);

//operatori relationali
friend bool operator < (const BigInt &, const BigInt &);
friend bool operator <= (const BigInt &, const BigInt &);
friend bool operator > (const BigInt &, const BigInt &);
friend bool operator >= (const BigInt &, const BigInt &);

//operatori de citire/scriere
friend ostream & operator <<(ostream &, const BigInt &);
friend istream & operator >>(istream &, BigInt &);
};

```

n nr. int

n sir

Definițiile funcțiilor

Constructori

Constructorul care creează un obiect de tip BigInt pe baza unui număr de tip int, transmis ca parametru:

```

BigInt::BigInt(int n)
{ //initializez cu 0 vectorul de cifre
    memset(cifre, 0, LGMAX);
    //extrag cifrele lui n si le memorez incepand cu unitatile
    lg=0; do {cifre[lg]=n%10; n/=10; lg++;} while(n);
}

```

Constructorul care creează un obiect de tip BigInt pe baza unui număr stocat într-un sir de caractere, transmis ca parametru:

```

BigInt:: BigInt(const char* s)
{ //initializez cu 0 vectorul de cifre
    memset(cifre, 0, LGMAX);
    lg=strlen(s); //determin lungimea
    if (lg>LGMAX) throw("OVERFLOW");
    if (lg)
        for (int i=lg-1; i>=0; i--)
            { if (s[i]>='0'&&s[i]<='9')
                cifre[lg-i-1]=s[i]-'0';
            else throw("INVALID NUMBER");
            }
        else lg=1; //sirul vid il transform in valoarea 0
}

```

Observați că, în cazul în care lungimea sirului este mai mare decât numărul maxim de cifre ale unui număr mare (LGMAX), „aruncăm” eroarea OVERFLOW, iar dacă sirul conține alte caractere decât cifrele zecimale, „aruncăm” eroarea INVALID NUMBER.

igInt &;

Constructorul de copiere

```
BigInt::BigInt(BigInt& ob)
{int i;
lg=ob.lg;
memset(cifre, 0, LGMAX);
for (i=0;i<lg;i++) cifre[i]=ob.cifre[i];
}
```

Functia e0()

Functia prietenă e0() verifică dacă numărul mare transmis ca parametru este nul:

```
bool e0(const BigInt & ob)
{ return ob.lg==1 && ob.cifre[0]==0; }
```

Operatorul de atribuire

```
BigInt& BigInt::operator = (const BigInt & ob)
{int i;
if (this != &ob) //nu este autoatribuire
{lg=ob.lg;
memset(cifre,0,LGMAX);
for (i=0; i<lg; i++) cifre[i]=ob.cifre[i];
}
return *this;
}
```

Operatorul de indexare

```
int BigInt::operator [] (const int poz) const
{ if (poz<0 || poz>LGMAX-1) throw("RANGE ERROR");
return cifre[poz]; }
```

Observați că operatorul de indexare „aruncă” o eroare în cazul în care indicele nu se află în intervalul [0, LGMAX).

Operatorii de incrementare/decrementare

```
BigInt& BigInt::operator++() //forma prefixata
{int i;
for (i=0; i<lg && cifre[i]==9; i++) cifre[i]=0;
if (i==LGMAX) throw("OVERFLOW");
cifre[i]++;
if (i==lg) lg++;
}
```

```
BigInt BigInt::operator++(int fictiv) // forma postfixata
{BigInt temp;
temp = *this;
++(*this);
return temp;
}
```

```

    BigInt& BigInt::operator--() //forma prefixata
    {int i;
     if (lg==1 && cifre[0]==0) throw ("UNDERFLOW");
     for (i=0; cifre[i]==0; i++) cifre[i]=9;
     cifre[i]--;
     if (lg>1 && cifre[lg-1]==0) lg--;
    }

    BigInt BigInt::operator--(int fictiv) // forma postfixata
    {BigInt temp;
     temp = *this;
     --(*this);
     return temp; }
```

În cazul în care numărul este nul, operatorul de decrementare „aruncă” eroarea UNDERFLOW.

Operatori de egalitate

```

bool operator == (const BigInt & ob1, const BigInt & ob2)
{int i;
 if (ob1.lg!=ob2.lg) return false;
 for (i=0; i<ob1.lg && ob1.cifre[i]==ob2.cifre[i]; i++);
 return i==ob1.lg; }

bool operator != (const BigInt & ob1, const BigInt & ob2)
{ return !(ob1==ob2); }
```

Operatori relationali

```

bool operator < (const BigInt & ob1, const BigInt & ob2)
{int i;
 if (ob1.lg < ob2.lg) return true;
 if (ob1.lg > ob2.lg) return false;
 for (i=ob1.lg-1; i>=0 && ob1.cifre[i]==ob2.cifre[i]; i--);
 if (i==-1) return false;
 if (ob1.cifre[i] < ob2.cifre[i]) return true;
 return false;
}

bool operator > (const BigInt & ob1, const BigInt & ob2)
{ return ob2<ob1; }

bool operator <= (const BigInt & ob1, const BigInt & ob2)
{ return !(ob1>ob2); }

bool operator >= (const BigInt & ob1, const BigInt & ob2)
{ return !(ob1<ob2); }
```

Operatori aritmetici și de atribuire compuși cu operatori aritmetici

Toți operatorii aritmetici sunt implementați utilizând operatorul de atribuire compus corespunzător.

```

BigInt& operator+=(BigInt &ob1, const BigInt &ob2)
{int t=0, s, i;
 if (ob2.lg > ob1.lg) ob1.lg=ob2.lg;
 for (i=0; i<ob1.lg; i++)
 {s=ob1.cifre[i]+ob2.cifre[i]+t;
  t=s/10;
  ob1.cifre[i]=s%10;
 }
 if (t)
 {if (ob1.lg==LGMAX) throw ("OVERFLOW");
  ob1.cifre[ob1.lg++]=t;}
 return ob1; }

BigInt operator+(const BigInt &ob1, const BigInt &ob2)
{BigInt temp;
 temp=ob1; temp+=ob2;
 return temp;
}

BigInt& operator-=(BigInt &ob1, const BigInt &ob2)
{int i, t=0, s;
 if (ob1<ob2) throw ("UNDERFLOW");
 for (i=0; i<ob1.lg; i++)
 {
  s=ob1.cifre[i]-ob2.cifre[i]+t;
  if (s<0) {s+=10; t=-1;}
  else t=0;
  ob1.cifre[i]=s;
 }
 while (ob1.lg>1 && ob1.cifre[ob1.lg-1]==0) ob1.lg--;
 return ob1;
}

BigInt operator-(const BigInt &ob1, const BigInt &ob2)
{BigInt temp;
 temp=ob1; temp-=ob2;
 return temp;
}

BigInt& operator*=(BigInt & ob1, const BigInt & ob2)
{int v[LGMAX];
 int i, j, t, s;
 memset(v, 0, sizeof(int)*LGMAX);
 if (e0(ob1) || e0(ob2))
 {ob1=BigInt();return ob1;} //rezultatul este 0
}

```

```

ci
l de atribuire
())
&ob2)
--;
&ob2)
ob2)
te 0

    for (i=0; i<ob1.lg; i++)
        for (j=0; j<ob2.lg; j++)
        {
            if (i+j>=LGMAX) throw ("OVERFLOW");
            v[i+j]+=ob1.cifre[i]*ob2.cifre[j];
        }
        for (t=i=0; i<LGMAX; i++)
        {
            s=t+v[i];
            v[i]=s%10;
            t=s/10;
        }
        if (t) throw ("OVERFLOW");
        //determin lungimea si copiez in ob1 cifrele
        for (ob1.lg=LGMAX; ob1.lg>1 && !v[ob1.lg-1]; ob1.lg--);
        for (i=0; i<ob1.lg; i++) ob1.cifre[i]=v[i];
        return ob1;
    }

    BigInt operator *(const BigInt & ob1, const BigInt & ob2)
    {
        BigInt temp;
        temp=ob1; temp*=ob2;
        return temp;
    }

    BigInt& operator /= (BigInt & ob1, const BigInt & ob2)
    {
        int cat[LGMAX], i, lgcat=0, cc;
        if (e0(ob2)) throw("Division by 0");
        if (ob1<ob2) { ob1=BigInt(); return ob1; } //catul este 0
        if (ob1==ob2){ ob1=BigInt(1); return ob1; } //catul este 1
        BigInt t;
        for (i=ob1.lg-1; t*10+ob1.cifre[i]<ob2; i--)
        {
            t*=10; t+=ob1.cifre[i];
        }
        //calculez cifrele catului
        for ( ; i>=0; i--)
        {
            t=t*10+ob1.cifre[i];
            for (cc=9; cc*ob2>t; cc--);
            t-=cc*ob2;
            cat[lgcat++]=cc;
        }
        ob1.lg=lgcat;
        for (i=0; i<lgcat; i++) ob1.cifre[i]=cat[lgcat-1-i];
        return ob1;
    }

    BigInt operator /(const BigInt & ob1, const BigInt & ob2)
    {
        BigInt temp;
        temp=ob1; temp/=ob2;
        return temp;
    }

    BigInt& operator %= (BigInt & ob1, const BigInt & ob2)

```

```

{int cat[LGMAX], i, lgcat=0, cc;
if (e0(ob2)) throw("Division by 0");
if (ob1<ob2) { return ob1; } //restul este ob1
if (ob1==ob2) { ob1=BigInt(); return ob1; } //restul este 0
BigInt t;
for (i=ob1.lg-1; t*10+ob1.cifre[i]<ob2; i--)
    { t*=10; t+=ob1.cifre[i]; }
//calculez cifrele catului
for ( ; i>=0; i--)
    {t=t*10+ob1.cifre[i];
     for (cc=9; cc*ob2>t; cc--);
     t-=cc*ob2;
     cat[lgcat++]=cc;
    }
while (t.lg>1 && t.cifre[t.lg-1]==0) t.lg--;
ob1=t;
return ob1; }

```

```

BigInt operator %(const BigInt & ob1, const BigInt & ob2)
{ BigInt temp;
 temp=ob1; temp%=ob2;
 return temp;
}

```

Operatorii de citire/scriere

La citire utilizăm un sir de caractere temporar, în care citim numărul, apoi reținem cifrele numărului în vectorul de cifre, în ordine inversă, transformând caracterele în cifre:

```

istream & operator >> (istream & in, BigInt & ob)
{char s[LGMAX+1];
in>>s;
ob.lg=strlen(s);
if (ob.lg>LGMAX) throw ("OVERFLOW");
for (int i=ob.lg-1; i>=0; i--)
    ob.cifre[ob.lg-i-1]=s[i]-'0';
return in;
}

```

Afișăm vectorul de cifre în ordine inversă:

```

ostream & operator << (ostream & out, const BigInt & ob)
{
for (int i=ob.lg-1; i>=0; i--)
    out<<(int)ob.cifre[i];
return out;
}

```

Problema 1. Descompunere Fibonacci

Considerăm sirul lui Fibonacci: $0, 1, 1, 2, 3, 5, 8, 13, 21, \dots$

Dat fiind un număr natural ($n \in \mathbb{N}$), scrieți acest număr sub formă de sumă de elemente neconsecutive din sirul Fibonacci, fiecare element putând să apară cel mult o dată, astfel încât numărul de termeni ai sumei să fie minim.

Date de intrare

Din fișierul de intrare **fib.in** se citește de pe prima linie numărul natural n . Acesta poate avea maximum 100 de cifre.

Date de ieșire

În fișierul de ieșire **fib.out** se vor afișa termeni ai sirului *Fibonacci*, câte unul pe linie, care respectă restricțiile din enunț.

Exemplu

Pentru $n=20$ fișierul de ieșire poate conține 2 5 13.

(Olimpiada Națională de Informatică, Constanța, 2000)

Soluție

Metoda utilizată este *Greedy*: la fiecare pas selectăm ca termen în sumă cel mai mare număr Fibonacci $\leq n$, apoi scădem din n numărul selectat. Procedeul se repetă cât timp $n > 0$.

Să demonstrăm că acest algoritm generează o soluție care respectă restricțiile din enunț. În primul rând, algoritmul se termină: la fiecare pas, n se micșorează, deoarece există întotdeauna un număr Fibonacci $\leq n$ (măcar 1), deci după un număr finit de pași va deveni nul.

În al doilea rând, niciodată nu vor fi selectați doi termeni consecutivi ai sirului Fibonacci. Să presupunem prin reducere la absurd că în sumă am selectat doi termeni consecutivi din sirul Fibonacci. Fie $fib[i+1]$ și $fib[i]$ cei mai mari termeni Fibonacci consecutivi selectați: $S = \dots + fib[i] + fib[i+1] + \dots$. Deoarece $fib[i] + fib[i+1] = fib[i+2]$, iar $fib[i+2] > fib[i+1] > fib[i]$, deducem că algoritmul Greedy va selecta $fib[i+2]$, nu pe $fib[i+1]$ (deoarece acesta este cel mai mare termen al sirului Fibonacci $\leq n$ la pasul respectiv).

Matematicianul belgian Edouard Zeckendorf a demonstrat prin inducție că scrierea unui număr natural ca sumă de termeni Fibonacci neconsecutivi este unică, făcând abstracție de ordinea termenilor (vezi Teorema lui Zeckendorf http://en.wikipedia.org/wiki/Zeckendorf's_theorem).

Pentru a implementa algoritmul Greedy descris, am utilizat o funcție `Generare_Fibonacci` care generează toți termenii Fibonacci $\leq n$ și îi memorează în vectorul `fib`. În funcția `main()` parcurgem vectorul `fib` de la sfârșit către început (astfel încât la fiecare pas să fim poziționați pe cel mai mare termen Fibonacci $\leq n$), îl selectăm în sumă (îl afișăm) și îl scădem din n .

Atât n, cât și vectorul fib sunt numere mari. Observați cât de elegantă, simplă și clară devine implementarea acestui algoritm, utilizând clasa BigInt și operatorii supraîncărcați de lucru cu numere mari:

```
#include <fstream>
#include "bigint.h"
#define NRMAX 10000
using namespace std;

ifstream fin("fib.in");
ofstream fout("fib.out");

BigInt n;
int nr;
BigInt fib[NRMAX];

void Generare_Fibonacci();

int main()
{ int poz;
fin>>n;
Generare_Fibonacci();
poz=nr-1;
while (!e0(n))
{//caut cel mai mare termen din sirul fibonacci <=n
    for ( ; n<fib[poz]; poz--);
    fout<<fib[poz]<<'\n';
    n-=fib[poz];
}
fout.close();
return 0;
}

void Generare_Fibonacci()
{ int i;
fib[0]=BigInt();
fib[1]=BigInt(1);
for (i=1; fib[i]<=n; i++)
    fib[i+1]=fib[i]+fib[i-1];
nr=i;
}
```

Problema 2. Pavaj

Aleea din fața școlii are lungimea L și trebuie să fie pavată cu dale de formă dreptunghiulară, având lățimea egală cu lățimea aleii, dar de diferite lungimi. Există n tipuri de dale, numerotate de la 1 la n, din fiecare tip existând un număr suficient de mare de exemplare.

Ce

De

Da

Fiș

separa

tipuri

separa

tipul i

Da

Fiș

singur

pavare

Rez

• 0

• 0

• 0

• di

• Rez

Exe

Sol

Voi

Sub

Să :

utilizâr

Rel

Pen

orice ti

de lung

nr [x

nr [c

Tin

mari.

#inc

#inc

ntă, simplă și
și operatorii

Cerință

Determinați în câte moduri se poate realiza pavarea aleii cu dalele existente.

Date de intrare

Fișierul de intrare pavaj.in conține pe prima linie două numere naturale separate printr-un spațiu: L n, unde L este lungimea aleii, iar n este numărul de tipuri de dale. Pe cea de-a doua linie se află n numere naturale nenule distințe separate prin câte un spațiu $d_1 \ d_2 \ \dots \ d_n$, unde d_i reprezintă lungimea dalelor de tipul i, pentru orice $i=1, n$.

Date de ieșire

Fișierul de ieșire pavaj.out va conține o singură linie pe care va fi scris un singur număr natural, reprezentând numărul de moduri în care se poate realiza pavarea aleii cu dalele existente.

Restricții și precizări

- $0 < L \leq 700$
- $0 < n \leq 250$
- $0 < d_i < 1000$, pentru orice $i=1, n$
- $d_i \neq d_j$, pentru orice $i=1, n$
- Rezultatul va avea maximum 300 de cifre.

Exemple

pavaj.in	pavaj.out
5 3	13
2 1 3	

(.campion 2006)

Soluție

Vom rezolva problema prin metoda programării dinamice.

Subproblemă

Să se determine $nr[x] =$ numărul de posibilități de a pava o aleă de lungime x, utilizând dalele existente.

Relația de recurență

Pentru a pava o aleă de lungime x începem prin a plasa o dală. Dacă poate fi de orice tip i ($i=1, n$). După plasarea unei dale de tipul i, mai rămâne de pavat o aleă de lungime $x-d[i]$ (desigur, dacă $x \geq d[i]$). Relația de recurență devine :

$$nr[x] = nr[x-d[1]] + nr[x-d[2]] + \dots + nr[x-d[n]] \quad (\text{dacă } x-d[i] \geq 0, i=1, n)$$

$$nr[0] = 1$$

Tinând cont de dimensiunea datelor de intrare, este necesară utilizarea numerelor mari.

```
#include <iostream>
#include "bigint.h"
```

dale de formă
lungimi. Există
număr suficient

```

#define NMAX 251
#define LMAX 701
using namespace std;
ifstream fin("pavaj.in");
ofstream fout("pavaj.out");

int n, L;
int d[NMAX];
BigInt nr[LMAX];

int main()
{ int i, x;
//citire
fin>>L>>n;
for (i=1; i<=n; i++) fin>>d[i];
//programare dinamica
nr[0]=BigInt(1);
for (x=1; x<=L; x++)
    for (i=1; i<=n; i++)
        if (d[i]<=x) nr[x]+=nr[x-d[i]];
//afisare
fout<<nr[L]<<'\n';
fout.close();
return 0;
}

```

Problema 3. SQR

Să considerăm n un număr natural și a_1, a_2, \dots, a_n o secvență de n valori naturale nenule.

Cerință

Să se scrie un program care să determine cel mai mic număr cu care ar putea fi înmulțit produsul $a_1 * a_2 * \dots * a_n$ astfel încât să se obțină un pătrat perfect nenul.

Date de intrare

Fișierul de intrare `sqr.in` conține pe prima linie numărul natural n . Pe cea de-a doua linie se află numerele naturale $a_1 a_2 \dots a_n$, separate prin câte un spațiu.

Date de ieșire

Fișierul de ieșire `sqr.out` va conține o singură linie pe care va fi scris un număr natural, care reprezintă numărul minim cu care ar putea fi înmulțit produsul $a_1 * a_2 * \dots * a_n$ astfel încât să se obțină un pătrat perfect nenul.

Resu
• 0 <
• 0 <
• Rez

Exe

Solu

Pent
trebuie
descom
produs
descom
O al
la putei
torul p
fiecare :
factorile
factori]
Cel
obține
impare.

```

#include <iostream>
#include <cmath>
#define NMAX 251
#define LMAX 701
using namespace std;
int fin>>n;
int d[NMAX];
int nr[LMAX];
int main()
{
    for (int i = 1; i <= n; i++)
        fin>>d[i];
    for (int x = 1; x <= LMAX; x++)
        for (int i = 1; i <= n; i++)
            if (d[i] <= x)
                nr[x] += nr[x - d[i]];
    cout << nr[LMAX] << endl;
    return 0;
}

```

Restrictii

- $0 < n \leq 1000$
- $0 < a_i \leq 100000$, pentru $1 \leq i \leq n$
- Rezultatul va avea maxim 1000 de cifre.

Exemple

sqr.in	sqr.out	Explicație
3	5	$12 * 3 * 5 = 180$
12		$180 * 5 = 900 = 30 * 30$
3		
5		

(campion 2005)

Soluție

Pentru ca un număr să fie patrat perfect nenul, descompunerea sa în factori primi trebuie să conțină doar factori la puteri pare. Prin urmare, trebuie să determinăm descompunerea în factori primi a produsului $a_1 * a_2 * \dots * a_n$. O idee ar fi să calculăm produsul, apoi să-l descompunem în factori primi (ceea ce ar fi cam complicat, descompunerea unui număr mare în factori primi nefiind foarte simplă).

O altă idee se obține plecând de la observația că dacă în a_1 apare factorul prim d la puterea p_1 , iar în a_2 factorul prim d apare la puterea p_2 , în produsul $a_1 * a_2$ factorul prim p apare la puterea $p_1 + p_2$. Prin urmare, este suficient să descompunem fiecare număr a_i ($1 \leq i \leq n$) în factori primi și să reținem într-un vector p suma puterilor factorilor primi (p[d] = suma puterilor factorului prim d din descompunerea în factori primi a numerelor a_i , $1 \leq i \leq n$).

Cel mai mic număr cu care poate fi înmulțit produsul $a_1 * a_2 * \dots * a_n$ pentru a obține un patrat perfect se determină înmulțind factorii primi care apar la puteri impare. Acesta va fi, evident, un număr mare.

```
#include <fstream>
#include "bigint.h"
#define NMAX 1001
#define DMAX 100000
using namespace std;
ifstream fin("sqr.in");
ofstream fout("sqr.out");

int n;
int a[NMAX];
int p[DMAX];
BigInt nr(1);

int main()
{int i, d;
//citire
fin>>n;
for (i=1; i<=n; i++) fin>>a[i];

```

```

//descompunerea in factori primi
for (i=1; i<=n; i++)
    for (d=2; a[i]>1; d++)
        while (a[i] % d == 0) { ++p[d]; a[i] /= d; }

//determin nr
for (d=2; d<DMAX; d++)
    if (p[d] % 2) nr *= d;

//afisare
fout << nr << '\n';
fout.close();
return 0;
}

```

2.20. Clasa string

Operațiile cu siruri de caractere în C necesită o bună cunoaștere a modului de reprezentare și o bună înțelegere a operațiilor cu pointeri. Din acest motiv, lucrul cu siruri în C („C-like”) are un nivel relativ înalt de dificultate. Din acest motiv, în biblioteca standard a limbajului C++ a fost implementată clasa `string`, pentru a facilita operațiile cu siruri de caractere. Pentru a utiliza această clasă, trebuie să includeti fișierul antet `string`:

```
| #include <string>
```

Constructorii clasei string

- `string ()`; - creează un sir vid.
 - `string (const string& s)`; - constructorul de copiere: creează un sir identic cu sirul s.
 - `string (const string & s, size_t poz, size_t nr = npos)`; - creează un sir care conține nr caractere din s, începând de la poziția poz. Dacă nr lipsește sau este prea mare, se copiază caracterele din s de la poziția poz până la sfârșitul sirului.
 - `string(const char* s)`; - creează un sir pe baza sirului de caractere din C s (secvență de caractere terminată cu caracterul 0).
 - `string(const char* s, size_t n)`; - creează un sir conținând primele n caractere din sirul de caractere s.
 - `string(size_t n, char c)`; - creează un sir care conține caracterul c repetat de n ori.

Observații

1. `size_t` este un tip care are ca valori numere naturale, capabil să rețină dimensiunea oricărui tip de date.
 2. `npos` este un membru static constant, care are valoarea maximă pentru tipul `size_t`. Când utilizăm această valoare ca lungime a unui subșir, are semnificația „până la sfârșitul sirului”.

Destructorul clasei string

`~string()` – eliberează memoria alocată dinamic de constructor.

Operatori supraîncărcați

- Operatorul de atribuire = este supraîncărat pentru a atribui unui sir de tip string un alt sir de tip string sau un sir de caractere din C ori un caracter:

```
string& operator= (const string& str);
string& operator= (const char* s);
string& operator= (char c);
```

- Operatorul + este supraîncărat ca funcție friend și poate fi utilizat pentru concatenarea a două siruri de tip string, a unui sir de tip string cu un sir de caractere din C, respectiv dintre un caracter și un sir de tip string:

```
string operator+ (const string& s1, const string& s2);
string operator+ (const string& s1, const char* s2);
string operator+ (const char* s1, const string& s2);
string operator+ (const string& s1, char s2);
string operator+ (char s1, const string& s2);
```

- Operatorul += este supraîncărat ca funcție membră și funcționează pentru concatenarea cu un sir string, un sir C și, respectiv, un caracter:

```
string& operator+= (const string& s);
string& operator+= (const char* s);
string& operator+= (char c);
```

- Operatorul de indexare [] returnează o referință la caracterul de pe poziția specificată poz:

```
char& operator[] (size_t poz);
const char& operator[] (size_t poz) const;
```

Dacă poz este egal cu lungimea sirului, va returna o referință la caracterul nul.
Dacă sirul este constant, atunci referința returnată va fi constantă.

- Operatorii de citire/scriere sunt supraîncărcați astfel încât să funcționeze ca pentru sirurile din C

```
istream& operator>> (istream& is, string& str);
ostream& operator<< (ostream& os, const string& str);
```

- Operatorii relationali și de egalitate sunt supraîncărcați ca funcții friend pentru a funcționa atât pentru compararea sirurilor de tip string, cât și pentru compararea unui sir de tip string cu un sir din C:

```
bool operator== (const string& s1, const string& s2);
bool operator== (const char* s1, const string& s2);
bool operator== (const string& s1, const char* s2);
bool operator!= (const string& s1, const string& s2);
```

```

bool operator!= (const char* s1, const string& s2);
bool operator!= (const string& s1, const char* s2);
bool operator< (const string& s1, const string& s2);
bool operator< (const char* s1, const string& s2);
bool operator< (const string& s1, const char* s2);
bool operator<= (const string& s1, const string& s2);
bool operator<= (const char* s1, const string& s2);
bool operator<= (const string& s1, const char* s2);
bool operator> (const string& s1, const string& s2);
bool operator> (const char* s1, const string& s2);
bool operator> (const string& s1, const char* s2);
bool operator>= (const string& s1, const string& s2);
bool operator>= (const char* s1, const string& s2);
bool operator>= (const string& s1, const char* s2);

```

Functii prietene supraîncărcate

- Funcția `getline()` citește caractere în sirul s până la întâlnirea caracterului delimitator d, respectiv până la întâlnirea marcajului de sfârșit de linie '\n'.

```

istream& getline (istream& in, string& s, char d);
istream& getline (istream& in, string& s);

```

- Funcția `swap()` interschimbă valorile a două siruri de tip `string`:

```
void swap (string& x, string& y);
```

Functii membre

Funcțiile membre sunt foarte numeroase, menționăm câteva dintre acestea. Lista tuturor funcțiilor membre poate fi consultată online în documentația limbajului C++ (<http://www.cplusplus.com>).

- Funcțiile `length()` și `size()` returnează lungimea sirului (numărul de caractere din sir):

```

size_t length() const;
size_t size() const;

```

- Funcția `empty()` returnează true dacă sirul este vid și false în caz contrar:

```
bool empty() const;
```

- Funcția membră `data()` returnează un sir de caractere din C (succesiune de caractere terminată cu caracterul nul), conținând caracterele din sirul curent:

```
const char* data() const;
```

- Funcția `push_back()` adaugă caracterul c la sfârșitul sirului curent:

```
void push_back (char c);
```

- Funcția `pop_back()` sterge ultimul caracter din sirul curent:

```
void pop_back();
```

- Funcția `insert()` inserează la poziția poz în sirul curent un sir s (care poate fi `string` sau din C):

```
string& insert (size_t poz, const string& s);
string& insert (size_t poz, const char* s);
```

Poate insera doar o subsecvență din sirul de tip `string` s (cea care începe la poz1 și are lungimea nr1):

```
string& insert (size_t poz, const string& s, size_t
poz1, size_t nr1);
```

Poate insera primele n caractere din sirul din C s:

```
string& insert (size_t poz, const char* s, size_t n);
```

Sau poate insera de n ori caracterul c:

```
string& insert (size_t poz, size_t n, char c);
```

- Funcția `erase()` șterge nr caractere din sirul curent începând de la poziția poz. Parametrul poz are valoarea implicită 0, iar nr are valoarea implicită npos (ștergerea se face până la sfârșitul sirului):

```
string& erase (size_t poz = 0, size_t nr = npos);
```

- Funcția `find()` caută în sirul curent un alt sir s (de tip `string` sau sir din C) începând de la poziția poz și returnează poziția de început a primei apariții (sau npos dacă nu apare):

```
size_t find (const string& s, size_t poz = 0) const;
size_t find (const char* s, size_t poz = 0) const;
```

Poate căuta doar primele n caractere din sirul din C s:

```
size_t find(const char* s, size_t poz, size_t n) const;
```

Sau poate căuta un caracter c:

```
size_t find (char c, size_t poz = 0) const;
```

- Funcția `replace()` înlocuiește în sirul curent lg caractere, începând de la poziția poz cu un alt sir s (de tip `string` sau din C):

```
string& replace(size_t poz, size_t lg, const string& s);
string& replace (size_t poz, size_t lg, const char* s);
```

sau doar cu un subșir al lui s (care începe la poziția poz1 și are lungimea lg1):

```
string& replace (size_t poz, size_t lg, const string& s,
size_t poz1, size_t lg1);
```

sau cu primele n caractere din s:

```
string& replace (size_t poz, size_t lg, const char* s,
size_t n);
```

sau cu n caractere c:

```
string& replace (size_t poz, size_t lg, size_t n, char c);
```

- Funcția `substr()` returnează un subșir al sirului curent, care începe la poziția poz și are lg caractere (valoarea implicită pentru parametrul lg este npos, ceea ce înseamnă că se consideră toate caracterele de la poziția poz până la sfârșitului sirului curent) :

```
string substr (size_t poz = 0, size_t lg = npos) const;
```

Funcții de conversie

- Funcțiile `stoi()`, `stol()`, `stoll()`, `stoul()`, `stoull()` convertesc un string s care conține un număr întreg scris în baza b (valoarea implicită pentru b fiind 10) și returnează numărul întreg obținut (de tip int, long, long long, unsigned long, respectiv unsigned long long). Dacă pointerul p este nenul, după apel în p se va afla adresa caracterului din sir care urmează după valoarea numerică convertită. Valoarea implicită pentru parametrul p este 0, caz în care p nu este utilizat.

```
int stoi (const string& s, size_t* p=0, int b=10);
long stol (const string& s, size_t* p=0, int b=10);
long long stoll (const string& s, size_t* p=0, int b=10);
unsigned long stoul(const string& s, size_t* p=0, int b=10);
unsigned long long stoull(const string& s, size_t* p=0, int b=10);
```

- Funcțiile `stof()`, `stod()` și `stold()` convertesc un string s într-un număr real și returnează numărul convertit ca rezultat (de tip float, double, respectiv long double). Semnificația parametrului p este aceeași ca la funcțiile de conversie în întreg :

```
long double stold (const string& s, size_t* p=0);
double stod (const string& s, size_t* p=0);
float stof (const string& s, size_t* p=0);
```

Problema 4. Ed

Să considerăm un text format numai din litere mici ale alfabetului englez. Asupra acestui text se pot executa următoarele operații de editare :

1. L (cursorul se mută cu o poziție la stânga ; în cazul în care cursorul se află la începutul textului, această operație nu are niciun efect).
2. R (cursorul se mută cu o poziție la dreapta ; în cazul în care cursorul se află la sfârșitul textului, această operație nu are niciun efect).
3. B (șterge caracterul din stânga cursorului ; în cazul în care cursorul se află la începutul textului, această operație nu are niciun efect).
4. I<secvența> (inserează secvența de litere mici specificată după litera I în poziția curentă a cursorului).
5. D<nr> (șterge începând cu poziția curentă a cursorului nr caractere ; în cazul în care numărul de caractere existente din poziția curentă a cursorului până la sfârșitul sirului este <nr, se vor șterge toate caracterele existente de la poziția curentă a cursorului până la sfârșitul sirului).

Inițial, cursorul de scriere se află la sfârșitul textului (după ultimul caracter).

Cerință

Scrieți un program care să aplique unui text dat o secvență de operații de editare și care să afișeze textul astfel obținut.

Date de intrare

Fișierul de intrare ed.in conține pe prima linie textul dat. Pe cea de-a doua linie este scris un număr natural N, care reprezintă numărul de operații de editare. Pe următoarele N linii sunt scrise cele N operații de editare (câte o operație pe o linie), în ordinea în care trebuie să fie executate.

Date de ieșire

Fișierul de ieșire ed.out va conține o singură linie pe care va fi scris textul obținut după executarea în ordine a celor N operații de editare.

Restricții și precizări

- Textul dat este format din cel mult 50000 litere mici ale alfabetului englez.
- $1 \leq N \leq 5000$
- La orice operație de inserare (I) se pot insera în text maximum 10 caractere.
- La orice operație de ștergere (D) se pot șterge din text maximum 1000 caractere.

Exemplu

ed.in	ed.out
qwerty	qwarbb
10	
L	
L	
D3	
L	
B	
R	
Ibb	
L	
Iaaa	
D2	

(.campion 2006)

Soluție

Soluția devine imediată, utilizând clasa string. Citim succesiv operațiile și executăm fiecare operație citită, în funcție de tipul ei:

```
#include <fstream>
#include <string>
#include <cstdlib>

using namespace std;
ifstream fin("ed.in");

```

```

ofstream fout("ed.out");

string s, op;
int poz;

int main()
{int i, N, lg;
fin>>s;
poz=s.length();
fin>>N;
for (i=0; i<N; i++)
{
    fin>>op;
    switch (op[0])
    {
        case 'L': {if (poz>0) poz--; break;}
        case 'R': {if (poz<s.length()) poz++; break;}
        case 'B': {if (poz>0) {s.erase(poz-1,1); poz--;} break;}
        case 'I': {s.insert(poz,op,1,string::npos); break;}
        case 'D': {lg=atoi(op.data()+1);
                     //lg=stoi(op.substr(1));
                     s.erase(poz,lg); poz-lg; break; }
    }
}
fout<<s<<'\n';
fout.close();
return 0;
}

```

Singura operație mai dificilă este ștergerea, deoarece a fost necesară conversia unui subșir al șirului op (în care este memorată operația curentă) într-un număr întreg (lg). Există două variante pentru a realiza acest lucru (cu funcția `stoi()`, care pentru compilatoare mai vechi nu funcționează, fiind introdusă în biblioteca limbajului în anul 2011) sau transformând cu ajutorul funcției `data()` subșirul extras în șir de caractere din C, apoi realizând conversia cu funcția `atoi()`.

2.21. Moștenirea

Moștenirea reprezintă unul dintre principiile fundamentale ale *POO* și, în esență, constă în posibilitatea ca dintr-un concept general să obținem un concept particular (sau mai specializat).

Procesul prin care se realizează moștenirea se numește *derivare*. Prin derivare, clasa derivată moștenește toate datele și funcțiile membre ale unei alte clase (denumită clasă de bază). Clasa derivată se diferențiază față de clasa de bază prin adăugarea unor membri (date sau funcții) sau prin supraîncărcarea funcțiilor membre ale clasei de bază.

Utilizând derivarea claselor, dezvoltarea aplicațiilor devine mai eficientă. Aplicațiile nu sunt de fiecare dată începute de la zero. O parte din cod este deja scrisă și va fi moștenită. Codul moștenit este deja testat, deci există o garanție a corectitudinii acestuia.

O clasă poate fi clasă de bază pentru mai multe clase derive, iar o clasă derivată poate fi clasă de bază pentru alte clase, obținându-se astfel *ierarhii de clase*. În unele dintre limbajele de programare orientată pe obiecte, orice clasă este o clasă derivată dintr-o altă clasă de bază, generală, definită în cadrul limbajului (de exemplu, în Java sau în C#, această clasă este denumită *Object*). Limbajul C++ fiind un limbaj hibrid, permite, așa cum am văzut deja, definirea unei clase fără derivarea acesteia dintr-o altă clasă.

Derivarea unei clase dintr-o clasă de bază

Sintaxa declarării unei clase derive (denumită ClasaD) dintr-o clasă de bază (denumită ClasaB) este :

```
class ClasaD: specifiant_acces ClasaB
{
    //declarații de date membre
    //declarații de funcții membre
};
```

În declarația clasei derive, specifiant_acces poate fi *public*, *protected* sau *private*. Dacă nu este menționat specifiantul de acces, el este considerat implicit *private*. Uzual, programatorii C++ utilizează la derivarea claselor specifiantul *public*.

Reamintim că membrii *private* ai unei clase pot fi accesăți doar în clasa respectivă și în funcțiile prietene ale clasei respective; membrii *protected* pot fi accesăți în plus și în clasele derive și în funcțiile prietene ale claselor derive.

Efectul specifiicatorilor de acces asupra membrilor clasei de bază este sintetizat în tabelul următor. Veți observa că, în clasa derivată, accesul este mai restrictiv decât în clasa de bază, în funcție de specifiantul de acces utilizat la derivare.

Atribut în clasaB	Specifiant acces	Acces în clasaD	Acces în exterior
public		private	inaccesibil
private	private	inaccesibil	inaccesibil
protected		private	inaccesibil
public		protected	inaccesibil
private	protected	inaccesibil	inaccesibil
protected		protected	inaccesibil
public		public	accesibil
private	public	inaccesibil	inaccesibil
protected		protected	inaccesibil

Exemplu

Să construim, într-o formă extrem de simplificată, o clasă pentru evidența resurselor umane la o companie. Orice persoană, indiferent că este client sau angajat, trebuie să aibă un nume și un cod unic de identificare în cadrul companiei.

Vom construi, în primă instanță, clasa Persoana astfel :

```
class Persoana
{
public:
    static int IdCrt;
protected:
    int Id;
    string Nume;
public:
    //constructori:
    Persoana();
    Persoana(const string& s);
    //destructor
    ~Persoana();
    //functii de acces
    int GetId();
    string GetNume();
    void SetNume(const string& );
    void Scrie();
};
```

Numerele de identificare ale persoanelor sunt generate automat, cu ajutorul datei membre statice `IdCrt`. Valoarea acesteia va fi incrementată de fiecare dată la crearea unei persoane (în funcțiile constructor). Am declarat doi constructori (unul fără parametri, care setează doar `Id`-ul, și unul cu un parametru de tip `string`, care setează și numele). În ambeii constructori și în destructor afișăm câte un mesaj, pentru a putea urmări modul în care se creează, respectiv se elimină obiecte.

Implementarea funcțiilor membre este :

```
Persoana::Persoana()
{ Id=++IdCrt; Nume="";
cout<<"Am creat o persoana fara nume\n"; }

Persoana::Persoana(const string& s)
{ Id=++IdCrt; Nume=s;
cout<<"Am creat o persoana cu nume\n"; }

//destructor
Persoana::~Persoana()
{ cout<<"Am eliminat o persoana\n"; }

//functii de acces
int Persoana::GetId() { return Id; }
string Persoana::GetNume() { return Nume; }
void Persoana::SetNume (const string& s) { Nume=s; }
void Persoana::Scrie() { cout<<Id<<' '<<Nume<<'\n'; }
```

Observați că în clasa Persoana am declarat datele membre utilizând specificatorul de acces `protected`. În acest mod avem acces direct la datele membre din clasele derivate. Acest lucru nu era obligatoriu. Am fi putut declara datele membre `private` și să le accesăm prin intermediul funcțiilor publice de acces.

Din clasa Persoana vom deriva clasa Angajat, care are în plus un salariu și o funcție (date membre) și funcții de acces la salariu și la funcție.

```
class Angajat : public Persoana
{
protected:
    string Functie;
    int Salariu;
public:
    int GetSalariu();
    void SetSalariu(int);
    string GetFunctie();
    void SetFunctie(string);
    void Scrie();
};
```

Definițiile funcțiilor membre fiind:

```
int Angajat::GetSalariu() { return Salariu; }
void Angajat::SetSalariu(int x) { Salariu=x; }
string Angajat::GetFunctie() { return Functie; }
void Angajat::SetFunctie(string s) { Functie = s; }
void Angajat::Scrie()
{ cout<<Id<<' '<<Nume<<' '<<Functie<<' '<<Salariu<<'\n'; }
```

Următoarea secvență de cod generează două obiecte de tip Persoana (x, utilizând constructorul implicit, și y, utilizând constructorul cu un parametru) și un obiect de tip Angajat (z). Deocamdată în clasa Angajat nu există un constructor definit, prin urmare, se apelează doar la constructorul implicit al clasei Persoana.

```
Persoana x, y("Popescu Vasile");
x.SetNume("Ionescu Dan");
x.Scrie();
y.Scrie();
Angajat z;
z.SetNume("Vasilescu Ana"); z.SetFunctie("contabil");
z.SetSalariu(1000);
z.Scrie();
```

După execuția acestei secvențe, pe ecran se va afișa:

```
Am creat o persoana fara nume
Am creat o persoana cu nume
1 Ionescu Dan
2 Popescu Vasile
Am creat o persoana fara nume
3 Vasilescu Ana contabil 1000
```

Am eliminat o persoana
 Am eliminat o persoana
 Am eliminat o persoana

Constructorii și destructorul în clasa derivată

Regulile de funcționare a constructorilor/destructorului rămân valabile și în clasele deriveate. Totuși există o serie de elemente specifice, care ilustrează relația dintre constructorii/destructorul clasei de bază și constructorii/destructorul clasei deriveate:

- pentru construirea unui obiect al clasei deriveate se apelează mai întâi constructorul clasei de bază, apoi se apelează constructorul clasei deriveate;
- la eliminarea unui obiect al unei clase deriveate se apelează mai întâi destructorul clasei deriveate, apoi destructorul clasei de bază.

Când declarăm un constructor în clasa derivată trebuie să specificăm în lista de parametri mai întâi parametrii pentru constructorul corespunzător din clasa de bază, iar în continuare cei pentru constructorul clasei deriveate.

Când definim constructorul clasei deriveate trebuie să specificăm și constructorul clasei de bază cu cei corespunzători, astfel:

```
clasaD :: clasaD(tip1 pb1, tip2 pb2, ..., tiph pbh, tipd1 pd1,  

tipd2 pd2, ..., tipdk pdk) : clasaB(pb1, pb2, ...pbh)
```

Definirea unui constructor în clasa derivată este obligatorie dacă în clasa de bază nu există un constructor fără parametri.

Pentru exemplificare, vom completa clasa *Angajat* cu trei constructori:

- unul fără parametri, care se bazează pe constructorul fără parametri al clasei de bază și construiește un angajat fără nume, funcție sau salarior;
- unul cu doi parametri, care se bazează pe constructorul fără parametri al clasei de bază și construiește un angajat fără nume, dar cu funcție și salarior;
- unul cu trei parametri (numele angajatului, funcția și salariul) care se bazează pe constructorul cu un parametru al clasei de bază (numele angajatului fiind transmis acestuia).

```
class Angajat : public Persoana
{
protected:
    string Functie;
    int Salariu;
public:
    int GetSalariu();
    void SetSalariu(int);
    string GetFunctie();
    void SetFunctie(string);
    void Scrie();
//constructori
    Angajat();
    Angajat(const string& sf, const int sal);
    Angajat(const string& sn, const string& sf, const int sal);
```

```
//destructor
~Angajat();
};
```

Definițiile pentru constructori și destructor sunt:

```
Angajat::Angajat(const string& sn, const string& sf, const
int sal): Persoana(sn)
{Functie=sf; Salariu=sal;
cout<<"Am creat un angajat cu nume, functie si salariu\n";
}

Angajat::Angajat(const string& sf, const int sal):
Persoana()
{Functie=sf; Salariu=sal;
cout<<"Am creat un angajat fara nume, ";
cout<<"cu functie si salariu\n"; }

Angajat::Angajat():Persoana()
{cout<<"Am creat un angajat fara functie si salariu\n"; }

//destructor
Angajat::~Angajat()
{ cout<<"Am eliminat un angajat\n"; }
```

Observați că atât constructorii, cât și destructorul afișează pe ecran mesaje specifice, astfel încât la execuție să putem identifica exact ce se execută.

Pentru exemplificare vom crea și vom afișa trei obiecte de tip Angajat (câte unul pentru fiecare constructor definit) și apoi le vom afișa.

La declararea unui obiect al clasei derivate trebuie precizată mai întâi lista parametrilor actuali pentru constructorul clasei de bază, apoi pentru constructorul clasei derivate.

```
Angajat x;
x.SetNume("Vasilescu Ana"); x.SetFunctie("contabil");
x.SetSalariu(1000);
x.Scrie();
Angajat y("director", 2000);
y.SetNume("Vasilescu Ana");
y.Scrie();
Angajat z("Popa Ion", "casier", 500);
z.Scrie();
```

Urmărind mesajele afișate pe ecran la execuția acestei secvențe de instrucțiuni, deducem că se apelează mai întâi constructorul clasei de bază, apoi constructorul clasei derivate. În cazul destructorilor este invers (se apelează mai întâi destructorul clasei derivate, apoi destructorul clasei de bază).

```
Am creat o persoana fara nume
Am creat un angajat fara functie si salariu
1 Vasilescu Ana contabil 1000
```

```

Am creat o persoana fara nume
Am creat un angajat fara nume cu functie si salariu
2 Vasilescu Ana director 2000
Am creat o persoana cu nume
Am creat un angajat cu nume, functie si salariu
3 Popa Ion casier 500
Am eliminat un angajat
Am eliminat o persoana
Am eliminat un angajat
Am eliminat o persoana
Am eliminat un angajat
Am eliminat o persoana

```

Moștenirea multiplă

În C++ este permisă moștenirea multiplă (o clasă poate fi derivată din mai multe clase de bază). În acest caz, la declararea clasei deriveate nu se specifică o singură clasă de bază, ci o succesiune de clase de bază, separate prin virgulă, fiecare clasă de bază fiind precedată de propriul specificator de acces. Toate regulile menționate pentru moștenirea simplă rămân valabile.

Există mulți programatori care consideră moștenirea multiplă inutilă (deoarece aceleași rezultate ar putea fi obținute și cu moștenire simplă), ba chiar un indiciu că ierarhia de clase nu este corect proiectată. Din limbajul Java, moștenirea multiplă a fost în mod deliberat eliminată.

Conversiile隐式 de tip

Compilatorul C++ admite următoarele conversii *implicite de tip*:

- dintr-un obiect al clasei deriveate într-un obiect al clasei de bază;
- dintr-un pointer (sau referință) către un obiect al clasei deriveate către un pointer (sau referință) către un obiect al clasei de bază.

Nu sunt permise conversiile inverse !

2.22. Includerea condiționată

Când clasaD este derivată din clasaB trebuie să includem fișierul antet cu declarația clasei de bază (clasaB.h) în fișierul antet al clasei deriveate. Acest lucru poate crea probleme, în sensul că într-o aplicație este posibil ca declarația unei clase să fie inclusă de mai multe ori, ceea ce ar genera eroare la compilare.

Pentru a evita astfel de situații, la începutul fișierului antet, pentru orice clasă, vom defini o constantă simbolică semnificativă, cu ajutorul directivei preprocesor #define :

```
| #define NUMECLASA_H
```

Pentru a evita ca un fișier antet să fie inclus de mai multe ori într-o aplicație, vom include condiționat conținutul acestuia, utilizând directiva preprocesor `#ifndef`.

```
#ifndef NUMECLASA_H
#define NUMECLASA_H
... //continutul fisierului antet numeclasa.h
#endif
```

2.23. Polimorfismul

Polimorfismul reprezintă un alt principiu fundamental al programării orientate pe obiecte. Cuvântul *polimorfism* descrie posibilitatea de a avea mai multe forme. În viață reală, polimorfismul este extrem de uzual. De exemplu, dacă aş spune „Deschide!”, această „funcție” va fi executată în mod diferit dacă este vorba de a deschide ușa sau a deschide ochii ori a deschide un fișier etc.

În C++, ideea este similară: apelul unei funcții membre va avea un efect diferit în funcție de obiectul care invocă funcția (funcția apelată este selectată în funcție de tipul obiectului).

În exemplul prezentat la secțiunea „Moștenire” am întâlnit deja polimorfismul. Atât în clasa de bază Persoana, cât și în clasa Angajat a fost definită funcția `Scrie()`. În clasa Persoana, funcția afișă doar Id și Nume, iar în clasa Angajat afișă în plus Functie și Salariu. De exemplu, să analizăm apelurile funcțiilor `Scrie()` din secvența următoare de cod :

```
Persoana x("Vasilescu Ana");
Angajat y("Popa Ion", "casier", 500);
x.Scrie();
y.Scrie();
```

Pe ecran se va afișa :

```
1 Vasilescu Ana
2 Popa Ion casier 500
```

deoarece primul apel al funcției `Scrie()` este pentru obiectul `x` de tip `Persoana`, al doilea este pentru obiectul `y` de tip `Angajat`.

Să analizăm acum următorul exemplu :

```
Angajat y("Popa Ion", "casier", 500);
Persoana &r=y;
r.Scrie();
Persoana *p;
p=&y;
p->Scrie();
```

Această secvență de cod va afișa pe ecran :

```
2 Popa Ion
2 Popa Ion
```

Observați că am declarat o referință `r` (către un obiect de tip `Persoana`) și un pointer `p` de asemenea către un obiect de tip `Persoana`. Atât `r`, cât și `p` primesc adresa obiectului `y` de tip `Angajat`. Ambele apeluri ale funcției `Scrie()` (deci și funcția apelată prin intermediul referinței `r`, și cea apelată prin intermediul pointerului `p`) produc același efect: se apelează funcția `Scrie()` din clasa de bază `Persoana`.

Acest mecanism este denumit *rezoluție statică* (*static resolution = static linkage = early binding*): compilatorul selectează funcția înainte de execuția programului, considerând la versiunea din clasa de bază.

Funcțiile virtuale

O funcție virtuală este o funcție din clasa de bază a cărei declarație este precedată de cuvântul-cheie `virtual`.

Definind în clasa de bază o funcție virtuală, iar în clasa derivată o funcție cu același nume, anunțăm compilatorul că nu dorim rezoluție statică pentru această funcție, ci dorim ca identificarea versiunii funcției ce trebuie să fie apelată să se realizeze în timpul execuției programului, în funcție de obiectul care o invocă. Acest mecanism se numește *rezoluție dinamică* (*dynamic resolution = dynamic linkage = late binding*).

Caracteristicile funcțiilor virtuale

- Sunt funcții membre nestatice.
- Dacă o funcție este declarată `virtual` în clasa de bază, pentru fiecare funcție din clasele derivate având același prototip se va face rezoluție dinamică.
- Suprăîncărcarea funcțiilor virtuale în clasele derivate nu este obligatorie.
- Dacă suprăîncărcăm funcția cu un alt prototip, este considerată o altă funcție, deci se face rezoluție statică.
- Constructorii nu pot fi funcții virtuale, deconstructorul poate fi funcție virtuală.
- Funcțiile inline nu pot fi virtuale.

Exemplu

Pentru exemplificare vom relua clasele `Persoana` și `Angajat` (cu o mică simplificare: eliminăm deconstructorul și mesajele afișate în constructori). În clasa `Persoana` vom declara funcția `Scrie()` ca funcție virtuală.

În plus, vom mai deriva din clasa `Persoana` încă o clasă denumită `Client`. În clasa `Client` adăugăm codul IBAN al clientului (ca dată membră), funcțiile de acces la codul IBAN și doi constructori.

Este necesar să utilizăm includere condiționată.

Fișierul antet `persoana.h`:

```
#ifndef PERSOANA_H
#define PERSOANA_H
#include <iostream>
#include <string>
using namespace std;

class Persoana
{ public:
```

```

    static int IdCrt;
protected:
    int Id;
    string Nume;
public:
    //constructori:
    Persoana();
    Persoana(const string& s);
    //functii de acces
    int GetId();
    string GetNume();
    void SetNume(const string& s);
    virtual void Scrie();
};

#endif

```

Observați că am declarat funcția `Scrie()` ca funcție virtuală, pentru a obține rezoluție dinamică. Fișierul `persoana.cpp`:

```

#include "persoana.h"
Persoana::Persoana()
{ Id=++IdCrt; Nume="" ; }
Persoana::Persoana(const string& s)
{ Id=++IdCrt; Nume=s; }
//functii de acces
int Persoana::GetId() { return Id; }
string Persoana::GetNume() { return Nume; }
void Persoana::SetNume (const string& s) { Nume=s; }
void Persoana::Scrie() { cout<<Id<< ' '<<Nume<<'\n'; }

```

Fișierul antet `angajat.h`:

```

#ifndef ANGAJAT_H
#define ANGAJAT_H
#include "persoana.h"

class Angajat : public Persoana
{ protected:
    string Functie;
    int Salariu;
public:
    int GetSalariu();
    void SetSalariu(int);
    string GetFunctie();
    void SetFunctie(string);
    void Scrie();
    Angajat();
    Angajat(const string&, const int);
    Angajat(const string&, const string&, const int);
};
#endif

```

Fișierul angajat.cpp:

```
#include <iostream>
#include "angajat.h"
int Angajat::GetSalariu() { return Salariu; }
void Angajat::SetSalariu(int x) { Salariu=x; }
string Angajat::GetFunctie() { return Functie; }
void Angajat::SetFunctie(string s) { Functie = s; }
void Angajat::Scrie()
{ cout<<Id<<' '<<Nume<<' '<<Functie<<' '<<Salariu<<'\n'; }
Angajat::Angajat(const string& sn, const string& sf, const int s): Persoana(sn)
{ Functie=sf; Salariu=s; }
Angajat::Angajat(const string& sf, const int s): Persoana()
{ Functie=sf; Salariu=s; }
Angajat::Angajat():Persoana()
{ Salariu=0; Functie=""; }
```

Fișierul antet client.h:

```
#ifndef CLIENT_H
#define CLIENT_H
#include "persoana.h"
class Client: public Persoana
{
protected:
    string CodIban;
public:
    string GetCodIban();
    void SetCodIban(const string& sc);
    void Scrie();
    //constructori
    Client();
    Client(const string& sn, const string& sc);
};
#endif
```

Fișierul client.cpp:

```
#include "client.h"
string Client::GetCodIban() { return CodIban; }
void Client::SetCodIban(const string& sc) { CodIban=sc; }
void Client::Scrie()
{ cout<<Id<<' '<<Nume<<' '<<CodIban<<'\n'; }
Client::Client():Persoana() { CodIban=""; }
Client::Client(const string& sn, const string& sc): Persoana(sn)
{ CodIban=sc; }
```

Fișierul care conține funcția main():

```
#include <iostream>
#include "angajat.h"
```

```

#include "client.h"
using namespace std;

int Persoana::IdCrt=0;
int main()
{Persoana x("Vasilescu Ana");
 Angajat y("Popa Ion", "casier", 500);
 Client z("Ionescu Dan", "RO49AAAA1B31007593840000");
 x.Scrie(); y.Scrie(); z.Scrie();
 Persoana *p;
 p=&x; p->Scrie();
 p=&y; p->Scrie();
 p=&z; p->Scrie();
 return 0;
}

```

Executând aplicația constituită din aceste fișiere, obținem pe ecran :

```

1 Vasilescu Ana
2 Popa Ion casier 500
3 Ionescu Dan RO49AAAA1B31007593840000
1 Vasilescu Ana
2 Popa Ion casier 500
3 Ionescu Dan RO49AAAA1B31007593840000

```

Observați că, în acest caz, compilatorul analizează conținutul pointerului, nu tipul acestuia, și se execută funcția `Scrie()` corespunzătoare.

Functiile virtuale pure

Dacă dorim să includem o funcție virtuală în clasa de bază și aceasta urmează să fie redefinită în clasele deriveate, dar nu există nicio definiție cu sens în clasa de bază, atunci putem defini o funcție virtuală pură.

Declarația unei funcții virtuale pure :

```

virtual tip nume_functie(lista_parametri)=0;

```

Construcția `=0` anunță compilatorul că nu există corpul funcției, că aceasta este o funcție virtuală pură.

2.24. Tratarea erorilor utilizând clasa exception

Biblioteca standard a limbajului C++ conține o clasă pentru tratarea erorilor. Această clasă se numește `exception` și, pentru a utiliza această clasă, trebuie să includem fișierul antet corespunzător :

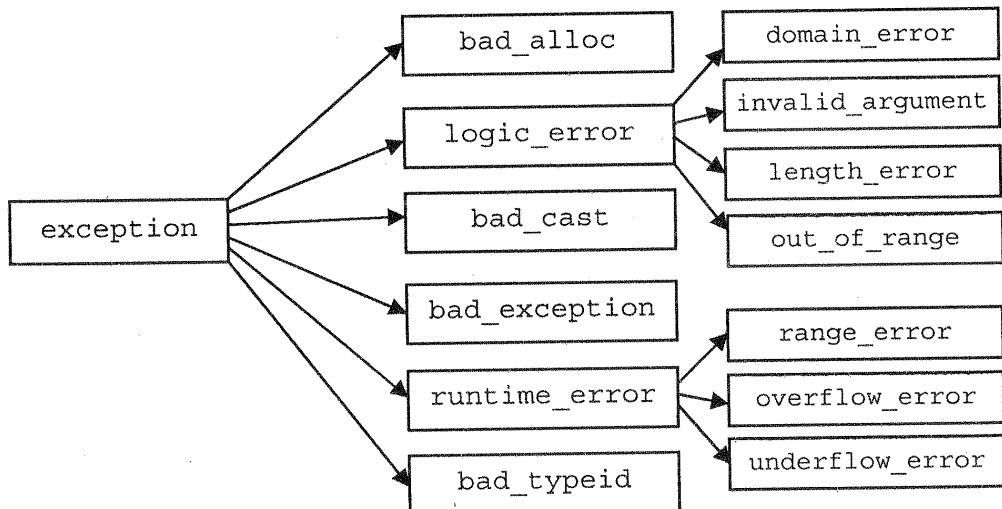
```

#include <exception>

```

Din clasa `exception` au fost deriveate clasele :

Numele clasei	Descriere
<code>exception</code>	Clasa de bază pentru toate excepțiile standard în C++.
<code>bad_alloc</code>	Excepții care pot fi generate la alocarea dinamică a memoriei (cu operatorul <code>new</code>) – se încearcă alocarea dinamică a unei zone de memorie, dar nu există suficientă memorie disponibilă.
<code>bad_cast</code>	Excepții generate de operatorul <code>dynamic_cast</code> , care apare atunci când se convertește un pointer sau o referință la un obiect al clasei de bază într-un pointer la un obiect al clasei derivate.
<code>bad_exception</code>	Excepțiile <code>bad_exception</code> sunt generate de erori neașteptate, care nu sunt capturate de niciun <code>catch</code> .
<code>bad_typeid</code>	Operatorul <code>typeid</code> returnează informații despre tipul unui obiect. O eroare <code>bad_typeid</code> este generată la încercarea de a aplica operatorul <code>typeid</code> unei expresii vide.
<code>logic_error</code>	Erori logice generate de program. Aceasta este clasa de bază pentru <code>domain_error</code> , <code>invalid_argument</code> , <code>length_error</code> , <code>out_of_range</code> .
<code>domain_error</code>	Excepție generată atunci când este utilizat un domeniu matematic invalid.
<code>invalid_argument</code>	Eroare generată de parametri incorecți.
<code>length_error</code>	Eroare generată la crearea unui obiect a cărui lungime depășește lungimea maximă admisă.
<code>out_of_range</code>	Eroare generată atunci când un indice este în afara domeniului (pentru clasa <code>vector</code> de exemplu, sau <code>bitset</code> , de către operatorul de indexare <code>[]</code>).
<code>runtime_error</code>	Erori generate în timpul execuției programului. Aceasta este clasa de bază pentru <code>overflow_error</code> , <code>underflow_error</code> , <code>range_error</code> .
<code>overflow_error</code>	O excepție generată de o eroare matematică de tip <code>overflow</code> (depășire superioară a domeniului de valori).
<code>range_error</code>	O excepție generată de încercarea de a memora o valoare care se află în afara domeniului de valori.
<code>underflow_error</code>	O excepție generată de o eroare matematică de tip <code>underflow</code> (depășire inferioară a domeniului de valori)



Funcția membră what ()

În clasa exception și în clasele derivate există o funcție membră denumită what (). Funcția returnează mesajul de eroare corespunzător unei erori apărute.

Exemplu de utilizare a excepției standard bad_alloc:

```
#include <iostream>
#include <exception>
using namespace std;
int main ()
{try
{ int* v= new int[1000000000]; }
catch (exception& e)
{ cout << "Standard exception: " << e.what() << '\n'; }
return 0; }
```

Pe ecran se va afișa :

```
Standard exception: std::bad_alloc
```

Exemplu de utilizare a excepției standard bad_typeid:

```
#include <iostream>
#include <exception>
#include <typeinfo>
using namespace std;
int main ()
{try
{ string* p=0;
typeid(*p);
//tentativa de a identifica tipul unui obiect nonexistent
}
catch (exception& e)
{ cout << "Standard exception: " << e.what() << '\n'; }
return 0;
}
```

„Aruncarea” explicită a unei erori standard

Funcția CitesteNenul() va „arunca” explicit o excepție de tip runtime_error în cazul în care se citește valoarea 0. Observați că pentru a putea utiliza clasa runtime_error am inclus stdexcept.

```
#include <iostream>
#include <stdexcept>
using namespace std;

int CitesteNenul()
{runtime_error eroare("Numar nul");
int val;
```

```

    cin >> val;
    if (val == 0) throw(eroare);
    return val;
}

int main ()
{
    try
    {
        int x=CitesteNenul();
        cout<<x;
    }
    catch (runtime_error& e)
    { cout<<"Eroare capturata: "<<e.what(); }
    return 0;
}

```

La introducerea valorii 0, programul va afișa pe ecran:

Eroare capturata: Numar nul

Derivarea propriei clase de excepții din clasa exception

În C++, pot fi „aruncate” erori de orice tip, dar acest lucru este imposibil în alte limbaje de programare. De exemplu, în Java sau în C# este posibilă numai „aruncarea” unei erori standard sau o eroare de un tip derivat din erorile standard. Uniformizarea modului de tratare a erorilor poate face programele mai ușor de întărit și de întreținut.

Exemplu

Vom deriva din clasa `logic_error` o clasă denumită `ExceptiaMea`. Declarația clasei:

```

#ifndef EXCEPTIAMEA_H
#define EXCEPTIAMEA_H
#include <stdexcept>
class ExceptiaMea : public std::logic_error
{public: ExceptiaMea(const char * s); };
#endif // EXCEPTIAMEA_H

```

Definiția constructorului:

```

ExceptiaMea :: ExceptiaMea (const char * s):
std::logic_error(s) { }

```

În funcția `CitesteNenul()` vom „arunca” o excepție de tip `ExceptiaMea` în cazul în care se citește valoarea 0:

```

#include <iostream>
#include "exceptiamea.h"
#include <stdexcept>
using namespace std;

```

```

int CitesteNenul()
{
    int val;
    cin >> val;
    if (val == 0) throw ExceptiaMea("Numar nul");
    return val;
}

int main ()
{
    try
    {
        int x=CitesteNenul();
        cout<<x;
    }
    catch (ExceptiaMea& e)
    { cout<<"Eroare capturata: "<<e.what(); }
    return 0;
}

```

2.25. Exerciții și probleme propuse

1. Un obiect este pentru o clasă ceea ce...
 - a. câine este pentru pisică.
 - b. filosof este pentru Kant.
 - c. Nichita Stănescu este pentru poet.
 - d. carte este pentru bibliotecă.
2. Presupunând că ați definit o clasă denumită Factura care conține data membră nestatică privată Cod, ce reține codul facturii. Considerând un obiect F de tip Factura, care dintre următoarele instrucțiuni afișează corect codul facturii ?

a. cout<<F.Cod;	b. cout<<F(Cod);
c. cout<<Factura.Cod;	d. Niciuna dintre variantele precedente.
3. Presupunând că în clasa Factura de la exercițiul precedent este definită o funcție membră nestatică publică denumită Total(), ce returnează suma totală de pe factura curentă. Considerând obiectul F de tip Factura, care dintre următoarele instrucțiuni afișează corect totalul de pe factură ?

a. cout<<F.Total();	b. cout<<Factura::Total();
c. cout<<Factura.Total();	d. Niciuna dintre variantele precedente.
4. Presupunând că în clasa Factura de la exercițiul precedent este definită o funcție membră nestatică publică denumită Scrie(), care afișează pe ecran date referitoare la factura curentă. În definiția acestei funcții în exteriorul clasei ne putem referi la data membră Cod astfel :

a. F.Cod	b. this->Cod
c. Cod	d. Factura.Cod

5. Considerând funcția `Scrie()` de la exercițiul precedent și că această funcție este definită în exteriorul clasei, care este antetul corect pentru definiția funcției `Scrie()` ?
- a. `Factura::void Scrie()` b. `void Factura.Scrie()`
c. `void Factura::Scrie()` d. `Factura::Scrie()`
6. Care dintre următoarele variante reprezintă o declarație corectă pentru un constructor al clasei `Factura` ?
- a. `void Factura();` b. `Factura()`
c. `void Factura::Factura();` d. `FacturaConstructor();`
7. Considerând că antetul constructorului clasei `Factura` este `Factura (int=1, string="Anonim")`, care dintre următoarele declarații sunt corecte ?
- a. `Factura F;` b. `Factura F(7, "Popa Vasile");`
c. `Factura F(7);` d. `Factura F("Popa Vasile");`
8. Care este declarația corectă pentru destructorul clasei `Factura` ?
- a. `~Factura(int);` b. `void ~Factura();`
c. `~Factura();` d. `~Factura(Factura&);`
9. Care dintre următorii constructori pentru clasa `Factura` pot coexista ?
- a. `Factura(int);` b. `Factura(int=0, string "");`
c. `Factura(int, string "");` d. `Factura();`
10. Care este declarația corectă a constructorului de copiere al clasei `Factura` ?
- a. `friend Factura(Factura);`
b. `Factura(const Factura&);`
c. `void Factura::Factura(const Factura&);`
d. `Factura& Factura (const Factura&);`
e. `Factura Factura(Factura&);`
f. Niciuna dintre variantele precedente.
11. Presupunând că am supraîncărcat operatorul binar + ca funcție prietenă a clasei `Factura`, care variantă poate fi un antet corect al definiției operatorului + ?
- a. `friend Factura operator+(Factura, Factura)`
b. `Factura operator+(const Factura&)`
c. `Factura operator+(const Factura&, const Factura&) const`
d. `Factura operator+(const Factura&, const Factura&)`
e. `Factura& operator+(const Factura&, const Factura&)`
f. Niciuna dintre variantele precedente.

12. Presupunând că supraîncărcăm ca funcție friend operatorul + și dorim să funcționeze și sub forma F1+F2+F3 (unde F1, F2, F3 sunt obiecte de tip Factura). În acest caz trebuie să :
- supraîncărcăm operatorul + pentru a accepta 3 parametri ;
 - returnăm ca rezultat un obiect de tip Factura ;
 - returnăm un rezultat având un tip standard al limbajului C++ ;
 - supraîncărcăm operatorul + fără parametri, astfel încât la apel să poată accepta un număr variabil de parametri ;
 - Niciuna dintre variantele precedente.
13. Presupunând că am supraîncărcat operatorul binar * ca funcție membră publică a clasei Factura, iar F1 și F2 sunt două obiecte de tip Factura, care dintre următoarele variante reprezintă o utilizare corectă a acestui operator ?
- F1*F2
 - operator*(F1, F2)
 - F1.operator*(F2)
 - Niciuna dintre variantele precedente
14. Pe orice factură trebuie să fie specificată valoarea taxei pe valoare adăugată (TVA). Care dintre următoarele variante reprezintă cea mai adekvată modalitate de a reține valoarea TVA-ului ?
- Dată membră nestatică ;
 - Dată membră statică ;
 - Variabilă globală ;
 - Variabilă locală funcției Scrie().
15. Presupunând că Firma este o dată membră statică de tip string a clasei Factura, care dintre următoarele afirmații sunt corecte ?
- Data membră Firma trebuie inițializată chiar de la declararea sa în cadrul declarației clasei Factura astfel : static string Firma = "" ;
 - În fișierul care conține funcția main(), inițializăm Firma ca variabilă globală astfel : string Factura::Firma = "VasileSRL" ;
 - În funcția main(), inițializăm Firma ca variabilă locală astfel : string Factura::Firma = "VasileSRL" ;
 - Declarăm un obiect F de tip Factura și apoi inițializăm Firma ca membru al lui F astfel : F.Firma = "VasileSRL" .
16. Presupunând că supraîncărcăm operatorul = pentru clasa Factura, care dintre următoarele variante reprezintă o declarație corectă pentru acest operator ?
- friend** Factura operator=(Factura&, const Factura&);
 - Factura& operator=(const Factura&);
 - Factura& operator=(Factura&, Factura&) const;
 - Factura& operator=(const Factura&) const;
 - void** operator=(Factura&, const Factura&);
 - Niciuna dintre variantele precedente.

17. Presupunând că supraîncărcăm operatorul << pentru a afișa o factură, care dintre următoarele variante reprezintă o declarație corectă a acestui operator?
- friend Factura operator<<(ostream&, const Factura&);**
 - friend ostream& operator<<(ostream&, const Factura&);**
 - friend void operator<<(const ostream&, const Factura&);**
 - ostream& operator<<(ostream&);**
 - ostream& operator<<(ostream&, Factura);**
 - Niciuna dintre variantele precedente.
18. Presupunând că dorim să supraîncărcăm pentru clasa Factura operatorul ++ ca funcție publică membră a clasei, atât în formă prefixată, cât și în formă postfixată. Care dintre următoarele afirmații sunt corecte?
- Trebuie să supraîncărcăm operatorul ++ de două ori, cu următorul prototip `Factura& operator ++();`
 - Trebuie să supraîncărcăm operatorul ++ o singură dată, cu următorul prototip `Factura& operator ++();`
 - Trebuie să supraîncărcăm operatorul ++ de două ori, cu următoarele prototipuri:
`Factura& operator ++();`
`Factura operator ++();`
 - Trebuie să supraîncărcăm operatorul ++ de două ori, cu următoarele prototipuri
`Factura& operator ++();`
`Factura operator ++(int);`
 - Operatorul ++ nu poate fi supraîncărcat pentru clasa Factura;
 - Niciuna dintre variantele precedente.
19. Principalul motiv pentru care utilizăm directivele preprocesor `#define`, `#ifndef`, `#endif` în fișierul antet al unei clase este că...
- dorim să atrăiem un nume semnificativ fișierului antet respectiv;
 - ne asigurăm că astfel clasa a fost declarată înainte de a fi utilizată;
 - astfel evităm declarația multiplă a unei clase în cadrul aceleiași aplicații;
 - realizăm o economie de memorie.
20. Prin supraîncărcare...
- este posibil să definim mai multe funcții cu aceeași listă de parametri, dar cu nume diferite;
 - să definim mai mulți destructori pentru o clasă;
 - să definim mai mulți constructori pentru o clasă;
 - să definim mai multe funcții cu același nume, dar care trebuie să difere prin tipul rezultatului;
 - să definim mai multe funcții cu același nume, dar care trebuie să difere prin lista parametrilor;
 - putem să utilizăm funcții cu un singur nume pentru a implementa toate operațiile necesare.

21. Care dintre următoarele perechi pot reprezenta o relație de tipul *clasa de bază/clasă derivată*?
- a. Punct/Poligon ;
 - b. Cilindru/Cerc ;
 - c. Cerc/Triunghi ;
 - d. Poligon/Poligon convex.
22. Care dintre următoarele variante reprezintă faptul că am derivat clasa X din clasa Y, cu specificatorul de acces public?
- a. class Y: public X
 - b. class Y: public class X
 - c. class X: public class Y
 - d. class X: public Y
23. Considerând că am derivat public clasa X din clasa Y, iar în clasa Y este declarată data membră d cu specificatorul de acces protected, care dintre următoarele afirmații sunt adevărate?
- a. Nu avem acces la d în clasa X.
 - b. Avem acces la d în orice funcție din X.
 - c. Avem acces la d în orice funcție prietenă cu clasa X.
 - d. Data membră d trebuie redeclarată în X pentru a avea acces la ea.
 - e. Avem acces la d numai în clasa Y și funcțiile prietene ale clasei Y.
 - f. Niciuna dintre variantele precedente.
24. Care dintre următoarele afirmații sunt corecte?
- a. Orice operator din limbajul C++ poate fi supraîncărcat.
 - b. Prin supraîncărcare putem modifica aritatea operatorilor.
 - c. Prin supraîncărcare prioritatea și asociativitatea operatorilor nu se schimbă.
 - d. Toți operatorii pot fi supraîncărcați atât ca funcții friend, cât și ca funcții membre ale clasei.
 - e. Pentru orice clasă creată trebuie să supraîncărcăm toți operatorii limbajului.
 - f. Unii operatori pot fi supraîncărcați atât ca operatori unari, cât și binari.
25. Să considerăm că A este clasă de bază pentru clasa B, iar B este clasă de bază pentru clasa C. Când creați un obiect al clasei C:
- a. Se apeleză mai întâi un constructor al clasei A, apoi un constructor al clasei B, apoi un constructor al clasei C.
 - b. Se apeleză mai întâi un constructor al clasei C, apoi un constructor al clasei B, apoi un constructor al clasei A.
 - c. Se apeleză doar un constructor al clasei C.
 - d. Este posibil să nu se apeleze niciun constructor, în cazul în care clasele nu au constructori definiți.
26. Definiți o clasă denumită Caine. Datele membre vor fi codul de identificare, numele, rasa, anul nașterii. Datele membre vor fi declarate private. Declarați funcții membre publice pentru acces la date și pentru afișare. Scrieți o aplicație în care să demonstrați că această clasă funcționează corect.

27. Complex

Un număr complex poate fi reprezentat în formă algebrică, reținând partea reală și partea imaginară a numărului. De exemplu, $z=2.5-3i$ este un număr complex (2.5 este partea reală, iar -3 este partea imaginară a numărului). Definiți o clasă denumită `Complex` pentru lucrul cu numere complexe. Datele membre vor fi partea reală și partea imaginară a numărului (ambele de tip `double`, ambele private). Definiți constructori, funcții de acces și operatorii specifici pentru lucrul cu numere complexe. Demonstrați într-o aplicație că această clasă funcționează.

28. Euclid

Utilizați clasa `BigInt` pentru a rezolva următoarea problemă.

Este bine-cunoscut algoritmul de calcul al celui mai mare divizor comun (`cmmdc`) cu algoritmul lui Euclid prin împărțiri repetitive. Conform acestui algoritm, `cmmdc` a două numere naturale nenule a și b se calculează păstrând restul împărțirii și reluând împărțirea cu vechiul împărțitor și vechiul rest. Algoritmul se va termina când restul împărțirii devine zero. Cel mai mare divizor comun al celor două numere a și b va fi ultimul împărțitor.

Pentru calculul celui mai mare divizor comun al perechii $(16, 22)$ se vor efectua succesiv împărțirile :

Deîmpărțit	Împărțitor	Rest	
16	22	16	Pasul 1
22	16	6	Pasul 2
16	6	4	Pasul 3
6	4	2	Pasul 4
4	2	0	Pasul 5

Vom numi un „pas” o operație de împărțire ce intervine în calculului `cmmdc`. Se observă că pentru determinarea $cmmdc(16, 22)=2$ au fost necesari 5 pași.

Cerință

Cunoscând valoarea unui număr natural n , scrieți un program care determină o pereche de numere naturale (a, b) mai mici sau egale cu n , al căror `cmmdc` se obține într-un număr maxim de pași. Dacă există mai multe perechi cu această proprietate, se va afișa cea minimă. Spunem că perechea (a, b) este mai mică decât perechea (x, y) , dacă $a < x$ sau $a = x$ și $b < y$.

Date de intrare

Fișierul de intrare `euclid.in` conține un singur număr natural n ($4 < n < 10^{200}$).

Date de ieșire

Fișierul de ieșire `euclid.out` va conține pe prima linie numărul maxim de pași determinat. A doua linie va conține un număr natural a reprezentând primul număr al perechii minime identificate, iar pe a treia linie se va scrie numărul b reprezentând al doilea număr din pereche.

Exemple

<i>euclid.in</i>	<i>euclid.out</i>	<i>Explicație</i>
8	5 5 8	Numărul maxim de pași pentru oricare pereche de numere mai mici egale cu 8 este 5. Perechea (5, 8) este minimă cu această proprietate.
12345678910	48 4807526976 7778742049	Numărul maxim de pași pentru oricare pereche de numere mai mici egale cu 12345678910 este 48. Perechea (4807526976, 7778742049) este minimă cu această proprietate.

(Lotul național de informatică pentru juniori, Iași, 2008)

29. *Replace*

Utilizați clasa *string* pentru a rezolva următoarea problemă.

Una dintre cele mai importante operații în timpul procesării unui text este operația *find&replace*, de găsire a unui sir și de înlocuire a acestuia cu un alt sir specificat. Vom considera pentru problema noastră un caz special: acela în care dorim să facem o serie de înlocuiră într-o linie de text, utilizând un set de reguli. Fiecare regulă specifică sirul care trebuie găsit, precum și sirul cu care se va înlocui acesta. Se pornește cu prima regulă, căutându-se sirul de înlocuit și înlocuindu-se cu cel precizat. Se continuă cu prima regulă până nu se mai poate aplica, deoarece sirul de înlocuit nu mai este găsit. Se procedează la fel cu celelalte reguli în ordinea în care sunt date. În plus, se precizează următoarele:

- pentru detectarea sirului de înlocuit se începe de fiecare dată de la începutul textului;
- după terminarea aplicării unei reguli nu se mai revine la ea;
- operațiile se desfășoară considerând literele mari diferite de literele mici.

Cerință

Date fiind un set de reguli și textul în care trebuie realizate înlocuirile, să se determine forma finală a textului.

Date de intrare

Fișierul de intrare *replace.in* conține pe prima linie numărul natural N ($0 \leq N \leq 10$) reprezentând numărul de reguli. Următoarele $2N$ linii conțin perechi de siruri, primul sir din pereche fiind sirul care se caută, iar pe rândul imediat următor sirul cu care trebuie înlocuit acesta. Ultimul rând al fișierului de intrare conține textul în care se vor face înlocuirile.

Date de ieșire

Fișierul de ieșire *replace.out* va conține o singură linie pe care va fi scris textul obținut după aplicarea celor N reguli de înlocuire.

Exemplu

replace.in	replace.out
3 esti este joi vineri azi esti joi si esti vesel	azi_este_vineri_si_este_vesel

(Olimpiada Municipală de Informatică, Iași, 2009)

30. Modificați clasa `BigInt` astfel încât să lucreze cu numere naturale scrise în baza b ($b \leq 36$). Un număr în baza b utilizează b cifre (cifrele mai mari decât 9 fiind codificate cu litere din alfabetul englez $A=10, B=11, \dots, Z=35$).
31. Derivați din clasa `BigInt` o clasă care să implementeze numerele întregi mari. Adăugați o dată membră care să reprezinte semnul numărului și supraîncărcați funcțiile și operatorii în mod corespunzător.
32. *Timp*

Creați o clasă denumită `Timp` pentru a memora timpul exprimat în ore (de la 0 la 23), minute (de la 0 la 59) și secunde (de la 0 la 59).

Creați un constructor cu trei parametri (pentru ore, minute, secunde), având toți valorile implicate 0. Constructorul trebuie să verifice faptul că restricțiile specificate pentru oră, minute, secunde sunt respectate.

Supraîncărcați operatorul de citire și operatorul de afișare pentru clasa `Timp` (timpul va fi afișat sub forma `oră:minut:secunde`, minutele și secundele având exact două cifre).

Supraîncărcați operatorul `+` pentru a aduna doi timpi. Dacă prin adunare se depășește `23:59:59`, se obține ca rezultat timpul corespunzător din ziua următoare.

Supraîncărcați operatorul `-` pentru a scădea doi timpi. Dacă scăzătorul este mai mare decât descăzutul, afișați timpul corespunzător din ziua precedentă.

Supraîncărcați și operatorii `+=`, respectiv `-=`.

Supraîncărcați operatorii de egalitate și relaționali.

Dentist

Creați o clasă denumită `ProgramareDentist`, în care să utilizați clasa `Timp` construită la problema precedență, pentru a rezolva următoarea problemă.

La un cabinet stomatologic, secretara realizează programarea pacienților. Pentru fiecare pacient programat, reține numele, timpul la care pacientul dorește să fie programat (ora și minutul), respectiv durata maximală estimativă a tratamentului (determinată pe baza vastei experiențe a secretarei și exprimată de asemenea în ore și minute). Doctorul lucrează după următorul algoritm:

- seosește la cabinet la ora la care este programat primul pacient ;
- nu lucrează mai mult de 8 ore ;
- când termină de tratat un pacient, iar durata efectivă a tratamentului este mai mică decât durata maximală estimată, doctorul se odihnește în timpul rămas până la durata maximală estimată ;
- secretara are suficientă experiență și estimează corect durata maximală a unui tratament, astfel că doctorul nu depășește niciodată durata maximală a tratamentului unui pacient ;
- după terminarea tratamentului unui pacient (și a eventualei perioade de odihnă), doctorul invită în cabinet următorul pacient programat ;
- doctorul nu suportă să existe pacienți în sala de așteptare ; fiecare pacient vine exact la ora la care este programat și intră imediat în cabinetul doctorului.

În fiecare zi, secretara analizează lista programărilor pentru a doua zi și anulează o serie de programări (telefonând pacienților pentru confirmare sau pentru reprogramare în altă zi), astfel încât :

- toate restricțiile din algoritmul de lucru al doctorului să fie respectate ;
- doctorul să trateze un număr maxim de pacienți.

Scriți un program care afișează lista programărilor confirmate pentru a doua zi.

34. Polinoame

Un polinom poate fi reprezentat ca un vector în care reținem coeficienții acestuia, pe poziția i în vector fiind plasat coeficientul monomului x^i . De exemplu, la polinomul de grad 5 $P(x) = x^5 - 4x^3 + 2x + 15$, vectorul de coeficienți va fi :

$P =$	15	2	0	-4	0	1
	0	1	2	3	4	5

Construiți o clasă denumită **Polinom** pentru lucrul cu polinoame având această reprezentare. Scrieți o aplicație în care să demonstrați funcționalitatea clasei.

35. Poligon convex

Creați clasa **Punct** pentru reprezentarea unui punct în plan specificat prin coordonatele sale carteziene. Creați clasa **Segment** (un segment fiind determinat de două puncte în plan). Utilizând clasele **Punct** și **Segment**, rezolvați următoarea problemă :

- În fișierul **convex.in** se află pe prima linie un număr natural n ($3 \leq n \leq 1000$) reprezentând numărul de vârfuri ale unui poligon convex. Pe următoarele n linii sunt scrise câte două numere reale reprezentând coordonatele vârfurilor poligonului (în ordinea abscisă, ordinată). Vârfurile poligonului sunt specificate în fișier în ordinea parcurgerii poligonului în sensul acelor de ceasornic.
- În fișierul **convex.out** afișați pe linii separate perimetru și aria poligonului convex citit din fișierul de intrare.

36. Sfere

Derivați din clasa `Punct` clasa `Punct3D` pentru reprezentarea unui punct în spațiu, specificat de asemenea prin coordonatele sale carteziene. Definiți clasa `Sferă` (o sferă fiind identificată prin coordonatele centrului său – un punct în spațiu – și rază). Utilizați clasele `Punct3D` și `Sferă` pentru următoarea problemă:

- Fișierul de intrare `sfere.in` conține pe prima linie un număr natural n ($2 \leq n \leq 100$), reprezentând numărul de sfere, iar pe următoarele n linii câte patru numere reale x y z r , reprezentând coordonatele centrului și raza pentru fiecare dintre cele n sfere. Scrieți un program care să verifice dacă există cel puțin două sfere care se intersectează. Programul va afișa pe ecran mesajul DA sau mesajul NU, după caz.

37. Date calendaristice

Creați o clasă pentru lucrul cu date calendaristice. Scrieți o aplicație în care să demonstrați că această clasă funcționează.

39. Biblioteca

Studiați activitatea de la biblioteca școlii și realizați o aplicație care să permită gestiunea unei biblioteci școlare. La bibliotecă trebuie să reținem :

- evidența cărților (pentru fiecare carte reținem cota cărții, titlul, autorul, editura, anul apariției) ; aplicația trebuie să permită adăugarea unei cărți, ștergerea unei cărți, căutarea cărților după titlu (căutare cu potrivire totală sau parțială), căutarea tuturor cărților unui autor etc. ;
- pentru cărțile de specialitate, rețineți domeniul cărții (informatică, drept, medicină etc.) ; aplicația trebuie să realizeze căutări după domeniu ;
- evidența cititorilor (pentru fiecare cititor reținem codul de identificare a cititorului, numele, adresa, telefonul, e-mail-ul) ; aplicația trebuie să permită adăugarea unui cititor, ștergerea unui cititor, căutarea cititorilor după nume (căutare cu potrivire totală sau parțială), căutarea unui cititor după codul de identificare etc. ;
- evidența împrumuturilor (un împrumut este identificat prin cota cărții împrumutate, codul cititorului care a împrumutat cartea, data la care a avut loc împrumutul și data la care a fost restituită cartea împrumutată ; utilizați pentru reprezentarea datelor calendaristice clasa definită la problema precedentă) ; aplicația trebuie să permită realizarea unui împrumut, restituirea unei cărți, vizualizarea tuturor împrumuturilor restante (cărțile pentru care termenul de predare a fost depășit, împreună cu datele cititorilor care au împrumutat aceste cărți), vizualizarea tuturor împrumuturilor unui cititor, vizualizarea tuturor cititorilor care au împrumutat o anumită carte etc. .

Într-o dată membră statică veți reține numărul de zile pentru care poate fi împrumutată o carte (același pentru toate cărțile și pentru toți cititorii).

39. *Teste grilă*

Construiți clasele necesare pentru a realiza o aplicație de lucru cu teste grilă. Un test grilă este format dintr-un număr variabil de itemi. Un item constă într-o întrebare pentru care se oferă exact patru variante de răspuns, dintre care doar un singur răspuns este corect. Când se generează un test grilă, se aleg în mod aleatoriu itemi distincți din lista itemilor disponibili, numărul acestora fiind prestatibil (dar fără a depăși 100). La rezolvarea unui test grilă, pentru evaluare vom considera că toți itemii din test au aceeași pondere pentru calculul punctajului la test. La evaluare trebuie să se afișeze punctajul total obținut la test, dar și feedback punctual pentru fiecare item. Feedbackul poate fi doar Corect/Incorect sau puteți afișa un feedback mai elaborat, specific itemului respectiv (pentru itemii care au asociat un feedback constructiv).

40. *Blackjack*

Blackjack, cunoscut și sub numele de 21, este unul dintre cele mai populare jocuri de cărți de casino din lume. Regulile jocului pot fi consultate pe Internet (de exemplu, vezi <http://ro.blackjack.org/reguli>). Construiți clasele necesare și realizați o aplicație pentru jocul de Blackjack.

41. *Catalogul electronic*

Construiți clasele necesare și proiectați o aplicație care să funcționeze ca un catalog școlar electronic. Catalogul electronic conține catalogul corespunzător fiecărei clase din școală. Catalogul unei clase conține toți elevii clasei, cu notele (inclusiv nota la teză, dacă este un obiect cu teză) și absențele (motivate și nemotivate) la fiecare obiect. Catalogul trebuie să permită toate operațiile permise în catalogul „oficial”.

3. Elemente de programare generică

Programarea generică este o paradigmă de programare ce are ca scop proiectarea componentelor *software* (algoritmi, structuri de date, mecanisme de alocare a memoriei etc.) într-un mod abstract (sau generic), astfel încât utilizarea acestora să fie posibilă în contexte cât mai variate. În acest scop, accentul este plasat pe organizarea acestor componente astfel încât să poată fi reutilizate și extinse în diverse situații și pe dezvoltarea algoritmilor și, prin urmare, să funcționeze eficient indiferent de tipurile/structurile de date implicate.

Programarea generică presupune un nivel înalt de abstractizare și are o complexitate ridicată, dar este o abordare naturală: indiferent că sortăm numere, oi sau elevi, metoda de sortare utilizată este practic aceeași.

Concret, pentru ca aceeași secvență de cod să poată funcționa pentru tipuri de date diferite, nu este necesar să o rescriem pentru fiecare tip de date, ci vom considera că tipul datelor este un parametru.

În C++, programarea generică este posibilă utilizând funcții și clase parametrizate după tip. Acestea sunt denumite funcții/clase şablon (*template*).

3.1. Funcțiile şablon

O *funcție şablon* este o funcție pentru care cel puțin unul dintre tipurile de date utilizate este parametrizat. Funcția şablon corespunde unei familii de funcții, care realizează aceeași prelucrare, dar asupra unor date de tipuri diferite.

Declararea/definiția unei funcții şablon este precedată de :

```
| template <class T1, class T2, ..., class Tk>
```

Ulterior, în antetul sau în corpul funcției, pot fi utilizate tipurile T₁, T₂, ..., T_k.

Exemplul 1. Minimul a două valori

De exemplu, vom scrie o funcție şablon pentru determinarea minimului a două valori. Vom prezenta un program complet, pentru a exemplifica modul de declarare, definire și apelul unei astfel de funcții.

```
| #include <iostream>
| #include <string>
```

```

using namespace std;

//declaratie
template <class T>
T minim(T, T);

int main()
{
    string s1("Ana are mere"), s2("Anatol");
    int i1=8, i2=5;
    double x1=3.14, x2=1.5;
    //apel
    cout<<"Min string: "<<minim(s1,s2)<<'\n';
    cout<<"Min int: "   <<minim(i1,i2)<<'\n';
    cout<<"Min double: "<<minim(x1,x2)<<'\n';
    return 0;
}
//definitie
template <class T>
T minim(T x, T y)
{
    if (x<y) return x;
    return y;
}

```

Funcția `minim()` poate fi apelată pentru orice tip de date `T` pentru care este definit operatorul `<`.

Exemplul 2. Afisarea a două valori de tipuri diferite

Pentru a exemplifica modul în care declarăm, definim și apelăm o funcție şablon care acceptă mai multe tipuri ca parametri, vom construi funcția `Scrie()`, care primește ca parametri două valori de tipuri (parametrizate) potențial diferite și afișează pe ecran valorile respective. Presupunem că în cazul tipurilor pentru care apelăm funcția există operatorul de afișare `<<` supraîncărcat.

```

#include <iostream>
#include <string>
using namespace std;
//declarare
template <class T1, class T2>
void Scrie(T1, T2);

int main()
{
    string s1("Ana are mere"), s2("Anatol");
    int i1=8, i2=5;
    double x1=3.14, x2=1.5;
    //apel
    Scrie(s1, i1); Scrie(x2, s2); Scrie(i1, x1);
    return 0;
}

```

```
//definire
template <class T1, class T2>
void Scrie(T1 a, T2 b)
{cout<<"Prima valoare: "<<a<<"      ";
 cout<<"A doua valoare: "<<b<<'\n';
}
```

Exemplul 3. Supraîncărcarea funcțiilor şablon

Funcțiile şablon pot fi supraîncărcate, ca orice alte funcții. Bineînțeles, la supraîncărcare, trebuie să ne asigurăm că funcțiile supraîncărcate diferă fără ambiguități prin lista parametrilor.

Pentru a exemplifica supraîncărcarea, definim funcția inv() o dată cu doi parametri (aceștia vor fi interschimbați) și o dată cu un singur parametru (acestuia i se va schimba semnul algebric).

```
#include <iostream>
#include <string>
using namespace std;

template <class T>
void inv(T&);

template <class T>
void inv(T&, T&);

int main()
{int i1=8, i2=5;
 double x1=3.14, x2=1.5;
 inv(i1); inv(i1, i2);
 inv(x2); inv(x1, x2);
 cout<<i1<<' '<<i2<<'\n';
 cout<<x1<<' '<<x2<<'\n';
 return 0;
}

template <class T>
void inv(T& x)
{ x=-x; }

template <class T>
void inv(T& a, T& b)
{ T aux=a; a=b; b=aux; }
```

3.2. Clasele şablon

O clasă *şablon* este o clasă pentru care cel puțin unul dintre tipurile de date utilizate este parametrizat. Clasa şablon corespunde practic unei familii de clase similare, care suportă aceleași prelucrări.

Din punct de vedere sintactic, declarația/definiția unei clase şablon este precedată de o construcție de tipul :

```
template <class T1, class T2, ..., class Tk>
```

În definiția clasei, evident, se utilizează tipurile T_1, T_2, \dots, T_k .

Pentru a utiliza o clasă şablon, trebuie să specificăm numele clasei, urmat de tipul actual încadrat între paranteze unghiu-lare (<>). În cazul în care clasa admite ca parametri mai multe tipuri, tipurile actuale vor fi specificate între paranteze unghiu-lare, separate prin virgulă.

Exemplu. Clasa şablon Stiva

Stiva este o structură de date abstractă, care suportă două operații :

- inserarea unui element la vârful stivei
- extragerea elementului din vârful stivei.

Deoarece accesul este limitat doar la elementul din vârful stivei, această structură de date funcționează după principiul **LIFO** (Last In First Out – ultimul intrat, primul ieșit).

Pentru exemplificare, vom defini clasa şablon **Stiva** care să implementeze acest tip de date abstract. Vom considera drept parametru tipul T al elementelor din stivă. Vom reprezenta o stivă ca un vector s cu maxim $SMAX$ elemente de tipul T ; data membră vf indică vârful stivei. Vom include 5 funcții membre publice : **Push()** (inserează un element în stivă, la vârf), **Pop()** (extrage elementul de la vârful stivei), **Top()** (accesează elementul de la vârful stivei), **Empty()** (verifică dacă stiva este vidă) și **Size()** (care returnează numărul de elemente din stivă).

```
#ifndef STIVA_H
#define STIVA_H
#include <stdexcept>
#define SMAX 10000
using namespace std;
template <typename T>
class Stiva
{protected: int vf; T s[SMAX];
public:
    Stiva() { vf=-1; }
    T& Top()
    { logic_error e("Stiva vida"); }
```

```

    if (vf<0) throw e;
    return s[vf];
}

T Pop()
{
    logic_error e("Stiva vida");
    if (vf<0) throw e;
    return s[vf--];
}

void Push (const T & x)
{
    logic_error e("Stiva plina");
    if (vf==SMAX-1) throw e;
    s[++vf]=x;
}

bool Empty() { return vf==0; }

int Size() { return vf+1; }

};

#endif // STIVA_H

```

Aplicație. Problema SL

Limbajul *SL* (*Stack Language*) lucrează cu o stivă (inițial vidă) și un singur registru (care inițial conține valoarea 0). Un program în limbajul *SL* este format dintr-o succesiune de instrucțiuni, câte o instrucțiune pe o linie, scrise cu majuscule. În program, instrucțiunile sunt numerotate de la 0. Executarea unui program începe cu prima instrucțiune a programului. Instrucțiunile limbajului *SL* sunt:

Instrucțiune	Efect	Exemplu
PUSH x	Introduce valoarea întreagă x în stivă.	Dacă stiva este: 1 2 și executăm PUSH 4, stiva va arăta astfel: 4 1 2
STORE	Valoarea de la vârful stivei este extrasă și este plasată în registru.	Dacă stiva este: 1 2 și executăm STORE, stiva va arăta astfel: 2 iar în registru se află valoarea 1.
LOAD	Conținutul registrului este copiat și introdus în stivă.	Dacă registrul are valoarea 4 și stiva este: 1 2 după executarea operației LOAD registrul va avea tot valoarea 4, iar stiva este: 4 1

PLUS	Extrage din stivă două valori (cele de la vârf), le adună și introduce în stivă rezultatul.	Dacă stiva este : 4 10 20 și executăm PLUS, stiva va arăta astfel : 14 20
TIMES	Extrage din stivă două valori (cele de la vârf), le înmulțește și introduce în stivă rezultatul.	Dacă am fi executat TIMES pentru aceeași stivă, am fi obținut : 40 20
IFZERO n	Dacă valoarea de la vârful stivei este 0, atunci se sare la cea de a n-a instrucțiune din program. În caz contrar, se continuă execuția cu instrucțiunea următoare.	
DONE	Afișează valoarea de la vârful stivei și încheie execuția programului (indiferent dacă mai există sau nu instrucțiuni).	

Cerință

Scriptă un program care să citească un program în limbajul SL și să îl execute.

Date de intrare

Fișierul de intrare sl.in conține pe prima linie un număr natural n reprezentând numărul de instrucțiuni din programul în limbajul SL care urmează. Următoarele n linii conțin cele n instrucțiuni ale programului, câte o instrucțiune pe o linie.

Date de ieșire

Fișierul de ieșire sl.out va conține o singură linie pe care va fi scrisă valoarea afișată de programul SL din fișierul de intrare.

Restricții

- $2 \leq n \leq 1000$
- În cazul instrucțiunilor PUSH și IFZERO există un singur spațiu între numele instrucțiunii și argument.
- Argumentul instrucțiunii PUSH este un număr întreg din intervalul $[-10000, 10000]$, iar pentru IFZERO un număr natural din $[0, 1000]$.
- Programul din fișierul de intrare este corect (nu va conține cicluri infinite, nu va extrage o valoare dintr-o stivă vidă).

Exemple

sl.in	sl.out	Explicație		
		Instrucțiune	Stiva	Registru
14	12	PUSH 5	5	0
PUSH 3		PUSH 3	3	0
PLUS			5	
STORE		PLUS	8	0
LOAD		STORE		8
IFZERO 11		LOAD	8	8
PUSH 4		IFZERO 11	8	8
LOAD		PUSH 4	4	8
PLUS			8	
DONE		LOAD	4	8
PUSH 10			8	
LOAD		PLUS	12	8
TIMES			8	
DONE		DONE	12	8
		PUSH 10		
		LOAD		
		TIMES		
		DONE		

(campion 2008)

Soluție

Vom citi instrucțiunile programului într-un vector `P` cu componente de tip `string`. Vom parcurge secvențial instrucțiunile programului și le vom executa. Pentru fiecare instrucțiune, extragem cuvântul-cheie al acesteia (căutând mai întâi caracterul spațiu cu funcția membră a clasei `string` `find()`, apoi extragem primul cuvânt cu funcția membră `substr()`). În funcție de instrucțiune, executăm operația corespunzătoare, utilizând operațiile implementate în clasa `Stiva`. Observați că în cazul instrucțiunilor `PUSH` și `IFZERO` a fost necesară și extragerea numărului și conversia sa în număr întreg.

```
#include <fstream>
#include <stdexcept>
#include "stiva.h"
#include <string>
#include <cstdlib>
#define NMAX 1000
using namespace std;
ifstream fin("sl.in");
```

```
ofstream fout("sl.out");

int n;
string P[NMAX];
Stiva<int> S;
int R=0;

void citire();
int executie();

int main()
{
    citire();
    fout<<executie()<<'\n';
    fout.close();
    return 0;
}

void citire()
{char c;
 fin>>n; fin.get(c);
 for (int i=0; i<n; i++) getline(fin,P[i],'\n');
}

int executie()
{int i, a, b, poz;
 string op, snr;
 for (i=0; i<n; i++)
    {poz=P[i].find(' ');
     op=P[i].substr(0,poz);
     if (op=="PUSH")
        {snr=P[i].substr(poz+1);
         S.Push(atoi(snr.data())); continue; }
     if (op=="STORE") { R=S.Pop(); continue; }
     if (op=="LOAD") { S.Push(R); continue; }
     if (op=="PLUS")
        { a=S.Pop(); b=S.Pop(); S.Push(a+b); continue; }
     if (op=="TIMES")
        { a=S.Pop(); b=S.Pop(); S.Push(a*b); continue; }
     if (op=="IFZERO")
        if (S.Top()==0)
           {snr=P[i].substr(poz+1);
            i=atoi(snr.data())-1; continue; }
     if (op=="DONE") { return S.Top(); }
    }
}
```

3.3. Exerciții și probleme propuse

1. Construirea unei funcții şablon este recomandată atunci când...
 - a. este necesară o funcție cu mai mulți parametri.
 - b. sunt necesare mai multe funcții cu același nume, dar care realizează prelucrări diferite, în funcție de parametri.
 - c. sunt necesare mai multe funcții, care au parametri de tipuri diferite, dar care realizează aceleași prelucrări.
 - d. este necesară crearea unei singure funcții, pentru a nu crea confuzii.
2. Care dintre următoarele variante precedă definiția unei clase şablon ?
 - a. template <T>
 - b. <template class T>
 - c. template class<T>
 - d. template <class T>
3. Presupunând că am definit clasa şablon X cu un singur tip ca parametru și considerând că dorim să declarăm un obiect ob, transmițând constructorului ca parametru valoarea 7, care dintre următoarele declarații sunt corecte ?
 - a. X ob(7);
 - b. X<int> ob=7;
 - c. X<int> ob(7);
 - d. X ob<int>(7);
4. Definiți o funcție şablon care să determine minimul dintr-un vector, indiferent de tipul componentelor vectorului. Apelați funcția creată pentru vectori cu elemente de tipuri diferite.
5. Definiți o funcție şablon care să sorteze un vector, indiferent de tipul componentelor vectorului. Apelați funcția creată pentru vectori cu elemente de tipuri diferite.
6. *Unific*

Utilizând clasa şablon Stiva, rezolvați următoarea problemă :

Se consideră un sir $A = (A_1, A_2, \dots, A_N)$, format din N numere naturale nenule. Două numere se consideră vecine dacă se află pe poziții alăturate (A_i are ca vecini pe A_{i-1} și A_{i+1} , pentru orice $1 < i < N$, A_1 are ca vecin doar pe A_2 , iar A_N are ca vecin doar pe A_{N-1}).

Dacă două elemente vecine A_i, A_{i+1} ($1 \leq i < N$) au cel puțin o cifră comună, ele se pot unifica. Procedeul de unificare constă în eliminarea din numerele A_i și A_{i+1} a tuturor cifrelor comune și adăugarea prin alipire a numărului obținut din A_{i+1} la numărul obținut din A_i , formându-se astfel un nou număr. Numărul A_i va fi înlocuit cu noul număr, iar numărul A_{i+1} va fi eliminat din sir. (De exemplu, numerele $A_i=23814$ și $A_{i+1}=40273$ au cifrele 2, 3, 4 comune, după unificare obținem $A_i=817$, iar A_{i+1} este eliminat ; observați că, dacă după eliminarea cifrelor comune numerele încep cu zerouri nesemnificative, acestea vor fi eliminate, apoi se realizează alipirea.) Dacă în urma eliminării cifrelor comune, unul dintre numere nu mai

are cifre, atunci numărul rezultat va avea cifrele rămase în celălalt. Dacă în urma eliminării cifrelor comune atât A_i cât și A_{i+1} nu mai au cifre, atunci ambele numere vor fi eliminate din sir, fără a fi înlocuite cu o altă valoare.

Ordinea în care se fac unificările în sir este importantă: la fiecare pas se alege prima pereche de elemente vecine A_i A_{i+1} care poate fi unificate, considerând sirul parcurs de la stânga la dreapta. (De exemplu, considerând $A_i=123$, $A_{i+1}=234$, $A_{i+2}=235$, se unifică A_i cu $A_{i+1} \Rightarrow A_i=14$, iar unificarea cu următorul număr nu mai este posibilă.)

Cerință

Cunoscându-se sirul celor N numere naturale, să se determine:

- cifra care apare cel mai frecvent în scrierea tuturor celor N numere; dacă există mai multe cifre cu aceeași frecvență de apariție maximă, se va reține cea mai mică cifră;
- sirul obținut prin efectuarea unui număr maxim de unificări, după regulile descrise în enunț.

Date de intrare

Fișierul de intrare `unific.in` conține pe prima linie o valoare naturală N, iar pe următoarele N linii, în ordine, cele N numere naturale din sirul A, câte un număr pe o linie.

Date de ieșire

Fișierul de ieșire `unific.out` va conține pe prima linie un număr natural c reprezentând cifra care apare cel mai frecvent în scrierea celor N numere naturale. Pe cea de-a doua linie, un număr natural Nr reprezentând numărul de numere naturale rămase în sir după efectuarea unui număr maxim de unificări. Pe cea de-a treia linie se vor scrie cele Nr numere naturale rămase, în ordinea din sir, separate prin câte un spațiu.

Dacă, în urma proiectului de unificare, toate numerele vor fi eliminate, fișierul de ieșire va conține o singură linie, pe care se va scrie cifra care apare cel mai frecvent în scrierea celor N numere naturale.

Restricții

- $1 \leq N \leq 100\ 000$
- Numerele din sirul inițial, precum și numerele obținute în urma unificărilor, nu vor depăși 10^{18}

Exemplu

(Olimpiada Județeană de Informatică 2013)

7. Construiți o clasă şablon denumită **Coadă**, care să implementeze această structură de date. Amintim că o coadă este o structură de date abstractă care permite următoarele operații :

- inserarea unui element la sfârșitul cozii
 - extragerea elementului de la începutul cozii.

Demonstrați funcționalitatea clasei într-o aplicație în care să utilizați cozi cu elemente de tipuri diferite.

4. STL. Concepte generale

4.1. Ce este STL ?

STL (Standard Template Library – bibliotecă de şabloane standard) este o bibliotecă reunind implementări generice a numeroase structuri de date și algoritmi fundamentali în informatică. *STL* nu este singura bibliotecă de acest tip, dar cu siguranță este cea mai utilizată, datorită modului în care a fost proiectată structura sa, a eficienței implementărilor și a faptului că programatorii pot integra cu ușurință componente *STL* în aplicațiile lor. Programatorii familiarizați cu structura *STL*, principiile *POO* și cele ale programării generice consideră această bibliotecă un instrument foarte eficient, care crește semnificativ productivitatea lor în dezvoltarea aplicațiilor.

Structura *STL* este aproape în întregime creația lui Alexander Stepanov, care a început să pună în practică primele sale idei referitoare la programarea generică în 1979 (același an în care Bjarne Stroustrup a început să elaboreze *C++*). Principalul colaborator al lui Alexander Stepanov a fost David Musser, care promova principiile programării generice încă de la începutul anilor '70. Inițial, biblioteca a fost concepută pentru limbajul *ADA*, dar nu s-a bucurat de succes în industria *software*. Ulterior, biblioteca a fost implementată în *C++*, datorită facilităților pe care acest limbaj le oferă pentru programarea generică eficientă. În 1992, Meng Lee a fost cooptat în proiect și, împreună, au reușit să finalizeze biblioteca *STL*, conform cerințelor comitetului pentru standardizarea ANSI/ISO a limbajului *C++*. La ora actuală, *STL* face parte din bibliotecile standard ale limbajului *C++* și producătorii de compilatoare includ *STL* în produsele pe care le distribuie.

STL conține clase container, algoritmi generici, iteratori, functori, adaptori și allocatori. Vom face o prezentare generală pentru fiecare categorie, urmând ca în capituloarele următoare să analizăm în detaliu clasele container. Nu ne propunem o abordare exhaustivă a bibliotecii *STL*. Dorim să realizăm însă o abordare operațională. Scopul nostru este formarea unei viziuni de ansamblu asupra bibliotecii *STL*, împreună cu o profundă înțelegere a conceptelor care stau la baza construirii acesteia, precum și formarea unor abilități de utilizare avizată a componentelor *STL* pentru rezolvarea unor probleme de natură algoritmică.

O documentație completă și foarte bine structurată pentru biblioteca *STL* poate fi consultată online la adresa <http://www.sgi.com/tech/stl>. Veți considera utilă și documentația *STL* inclusă în documentația pentru limbajul *C++* la adresa <http://www.cplusplus.com/reference/stl>.

4.2. Clasele container

Un *container* este un obiect care conține o colecție de alte obiecte, denumite elementele containerului. În funcție de modul de organizare a elementelor, în *STL* există două categorii de clase container:

- containere secvențiale (*sequence containers*) – conțin o succesiune de elemente de același tip *T*; containerele secvențiale din *STL* sunt *vector*, *deque* și *list*; în *C++ 11* au fost introduse încă două containere secvențiale: *array* și *forward_list*;
- containere asociative (*associative containers*) – permit accesul rapid la elemente, prin intermediul unor chei; containerele asociative sortate din *STL* sunt *set*, *multiset*, *map* și *multimap*; în *C++ 11* au fost introduse și containerele asociative nesortate: *unordered_set*, *unordered_multiset*, *unordered_map* și *unordered_multimap*.

Practic, containerele implementează diferite structuri de date. În funcție de problema pe care trebuie să o rezolvați pentru a obține o implementare eficientă, trebuie să alegeti clasa container adecvată.

Utilizarea claselor container

Clasele container dispun, în mod unitar, de o serie de funcții membre și operatori supraîncărcați. În funcție de container, evident, există și alte funcții și operatori disponibili, specifice clasei. Prezentăm în continuare câteva exemple.

Tipurile definite în clase container

Orice clasă container din *STL* definește în mod public următoarele tipuri:

Tip	Semnificație
<i>value_type</i>	Tipul elementelor containerului
<i>pointer</i>	Pointer cu tipul de bază <i>value_type</i> (<i>T*</i>) – conține adresa unui element al containerului
<i>reference</i>	Referință cu tipul de bază <i>value_type</i> (<i>T&</i>)
<i>iterator</i>	Tipul iteratorului
<i>difference_type</i>	Tip întreg, reprezentând tipul rezultatului diferenței dintre doi iteratori
<i>size_type</i>	Tip întreg pozitiv, care reprezintă tipul dimensiunii unei zone de memorie
<i>reverse_iterator</i>	Tipul iteratorului invers

Tipurile *pointer*, *reference*, *iterator* și *reverse_iterator* au definite și versiunile constante (*const_pointer*, *const_reference*, *const_iterator* și *const_reverse_iterator*). Versiunile constante nu permit modificarea elementului indicat. Existența versiunilor constante nu este doar o măsură de securitate, ci o necesitate, deoarece în funcții *const* pot fi utilizate doar versiuni *const*.

Constructorii și destructorul

Toate clasele container au un constructor implicit (constructor fără parametri), un constructor bazat pe un domeniu (primește ca parametri prim – care indică primul element al domeniului – și ultim – care indică poziția de după ultimul element al domeniului; acest constructor creează un container și îl initializează cu obiectele din domeniul [prim, ultim]), un constructor de copiere și un destructor.

Operatorul de atribuire

Pentru toate clasele container este supraîncărcat operatorul de atribuire =.

Funcțiile membre referitoare la dimensiunea containerului

Clasele container conțin trei funcții membre care se referă la dimensiune:

- size() – returnează numărul actual de elemente din container;
- empty() – returnează o valoare de tip bool (true pentru un container vid, adică un container pentru care size() == 0 și false în caz contrar);
- max_size() – returnează numărul maxim de elemente ce pot fi stocate în container.

Interschimbarea a două containere

Interschimbarea a două containere se poate realiza prin intermediul funcției swap() care poate fi definită atât ca funcție membră, cât și ca funcție nemembră (algoritm). Fie c1 și c2 două containere. Interschimbarea lor prin intermediul funcției swap() se realizează astfel:

```
c1.swap(c2); //functia membră swap()
swap(c1,c2); //functia nemembra swap()
```

Inserarea elementelor într-un container

Clasele container dispun de funcția membră insert(), care permite inserarea unui element într-un container sau a unui domeniu de elemente într-un container.

Eliminarea elementelor dintr-un container

Funcția membră erase() permite eliminarea unui element specificat dintr-un container sau a elementelor dintr-un domeniu specificat. Există și funcția membră clear(), care determină ștergerea tuturor elementelor din container.

4.3. Clasele adaptor pentru containere

Clasele adaptor pentru containere sunt definite pe baza unei alte clase container, asigurând o funcționalitate restrânsă pentru aceasta. Există trei clase adaptor predefinite în STL, bazate pe containere secvențiale: stack (stiva), queue (coada) și priority_queue (coada cu prioritate).

4.4. Iteratorii

Iteratorii permit accesul la elementele unui container, independent de modul în care aceste elemente sunt stocate.

Iteratorii sunt asemănători cu pointerii. Mai exact, iteratorii sunt o generalizare a pointerilor, fiind obiecte care indică (*point*) alte obiecte.

Fiecare clasă container are definiți iteratorii proprii. Declarația unui iterator *it* pentru clasa container *CC<T>* este :

CC<T>::iterator it;

(*it* permite accesul la elementele containerului *CC<T>* pentru citire și scriere)

CC<T>:: const_iterator it;

(*it* permite accesul la elementele containerului *CC<T>* pentru doar pentru citire)

O declarare schematică a clasei *Iterator* este :

```
template<class T>
class Iterator
{
public:
    ...           // constructori, destructor
    bool operator==(const Iterator<T>&) const;
    bool operator!=(const Iterator<T>&) const;
    Iterator<T>& operator++();      // prefixat
    Iterator<T> operator++(int);   // postfixat
    T& operator*() const;
    T* operator->() const;
private: ...
};
```

Din schema precedentă deducem că orice iterator permite următoarele operații :

- comparații (cu ajutorul operatorilor supraîncărcați == și !=);
- incrementare (utilizând operatorul supraîncărcat ++, atât în formă prefixată, cât și în formă postfixată);
- dereferențiere (cu ajutorul operatorului de dereferențiere * poate fi accesat elementul indicat de un iterator, asemănător cu dereferențierea pointerilor);
- selecție indirectă (cu ajutorul operatorul de selecție indirectă -> se poate selecta un membru al obiectului indicat de iterator, prin intermediul iteratorului; funcționalitatea acestui operator este similară cu cea din cazul pointerilor).

Clașele container conțin funcții membre necesare iteratorilor pentru parcurgerea elementelor :

- funcția *begin()* returnează un iterator care indică primul element al containerului;
- funcția *end()* returnează un iterator care indică poziția de *după* ultimul element al containerului.

Observații

1. Domeniul unui iterator poate fi definit cu ajutorul funcțiilor `begin()` și `end()` astfel: `[begin(), end()]`.
2. Domeniul unui iterator este vid (containerul nu conține elemente) dacă `begin() == end()`.
3. Pentru a parcurge elementele unui container `Cob` de tip `CC<T>`, utilizăm un iterator `it`, pe care îl inițializăm cu ajutorul funcției `begin()`; cât timp nu am parcurs toate elementele (`it != end()`), iteratorul este incrementat:

```
CC<T> Cob;
CC<T>::iterator it;
for (it=Cob.begin(); it!= Cob.end(); ++it) ...
```

În funcție de operațiile ce pot fi efectuate cu iteratorii, în *STL* există cinci categorii de iteratori. Tabelul următor ilustrează pentru fiecare categorie de iteratori:

- modul în care iteratorul poate accesa elementele containerului (pentru citire: `*it` sau `it->membru`, fără a fi utilizat în membrul stâng al unei atribuirii; pentru scriere: `*it=valoare`, adică utilizat doar în membrul stâng al unei atribuirii; respectiv pentru citire și scriere);
- sensul în care se poate realiza parcurgerea elementelor containerului (doar înainte cu ajutorul operatorului `++`, în ambele sensuri cu operatorii `++` și `--`, respectiv dacă este permis accesul direct [aleatoriu] la elementele containerului).

Iteratori	Operații permise	Clase asociate
Iteratori de intrare (<i>input iterators</i>)	Acces pentru citire Parcuregere înainte	<i>istream</i>
Iteratori de ieșire (<i>output iterators</i>)	Acces pentru scriere Parcuregere înainte	<i>ostream, inserter</i>
Iteratori de înaintare (<i>forward iterators</i>)	Acces pentru citire și scriere Parcuregere înainte	
Iteratori bidirecționali (<i>bidirectional iterators</i>)	Acces pentru citire și scriere Parcuregere în ambele sensuri	<i>list, set, multiset, map, multimap</i>
Iteratori cu acces aleatoriu (<i>random access iterators</i>)	Acces pentru citire și scriere Acces direct (aleatoriu)	<i>vector, deque, string, array</i>

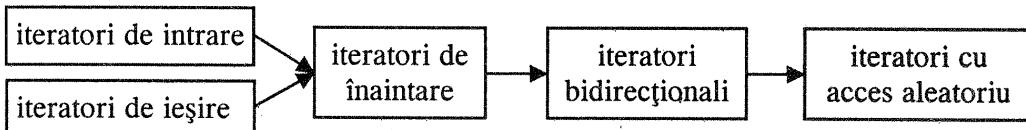
Observații

1. Iteratorii de intrare permit citirea datelor dintr-o secvență (printr-o singură parcurgere), în timp ce iteratorii de ieșire permit doar scrierea valorilor într-o secvență (de asemenea printr-o singură parcurgere).
2. Iteratorii de înaintare permit atât accesul la elementele containerului pentru citire, cât și pentru scriere; permit parcurgerea unei secvențe doar într-un singur sens, însă, spre deosebire de iteratorii de intrare/ieșire, elementele containerului pot fi parcuse de mai multe ori. Iteratorii bidirecționali sunt similari cu cei de înaintare, numai că permit în plus parcurgerea secvenței în sens opus (cu operatorul `--`).

3. Iteratorii cu acces aleatoriu suportă toate operațiile permise pentru operatorii bidirecționali ; în plus, sunt supraîncărcați operatorii care permit accesul direct la elementele containerului :

Operator	Sintaxă	Efect
Indexare	it [n]	Acces la elementul de pe poziția n
Adunare cu întreg	it + n n + it	Iterator care indică al n-lea element următor (pentru n pozitiv) sau precedent (pentru n negativ)
Scădere cu întreg	it - n	Iterator care indică al n-lea element precedent (pentru n pozitiv) sau următor (pentru n negativ)
Atribuire compus cu adunare	it += n	Iteratorul it va indica al n-lea element următor (pentru n pozitiv) sau precedent (pentru n negativ)
Atribuire compus cu scădere	it -= n	Iterator it va indica al n-lea element precedent (pentru n pozitiv) sau următor (pentru n negativ)
Scăderea a doi iteratori	it1 - it2	Returnează un număr întreg reprezentând distanța dintre doi iteratori (numărul de elemente din domeniul determinat de cei doi iteratori)
Operatori relaționali	it1 < it2 it1 > it2 it1 <= it it1 >= it2	Returnează o valoare bool indicând dacă elementul indicat de it1 precedă elementul indicat de it2 (<); efect similar pentru >, >=, <=.

4. Analizând operațiile disponibile pentru fiecare categorie de iteratori, deducem o ierarhizare a iteratorilor :



Orice iterator cu acces aleatoriu este și iterator bidirectional ; orice iterator bidirectional este și iterator de ținare ; orice iterator de ținare este iterator de intrare/ieșire.

4.5. Clasele adaptor pentru iteratori

O clasă adaptor pentru iteratori este o clasă şablon care schimbă modul de funcționare a unui iterator.

Iteratorul invers (reverse_iterator)

În STL este definit adaptorul `reverse_iterator` (iterator invers), care transformă un iterator bidirectional sau un iterator cu acces aleatoriu într-un iterator care permite parcurgerea elementelor unui container în ordine inversă.

Pentru a putea fi parcuse cu un iterator invers, clasele container conțin două funcții membre similare cu `begin()` și `end()`. Acestea sunt:

- `rbegin()` - returnează un iterator care indică ultimul element al containerului (aceasta este poziția de la care începe o parcurgere inversă);
- `rend()` - returnează un iterator care indică poziția *dinainte* de primul element al containerului (aici se termină o parcurgere inversă).

Pentru un `reverse_iterator`, operatorii de incrementare și decrementare se comportă invers. Prin urmare, iată cum se implementează parcurgerea în ordine inversă a elementelor unui container `Cob` de tip `CC<T>` cu ajutorul iteratorului invers `rit`:

```
CC<T> Cob;
CC<T>::reverse_iterator rit;
for (rit=Cob.rbegin(); rit!= Cob.rend(); ++rit) ...
```

Iteratorul de inserare

Iteratorii de inserare sunt adaptori care transformă un iterator de ieșire astfel încât la o atribuire de tipul `*it=valoare` nu va fi suprascris un element al containerului, ci este inserată în container valoarea specificată. O altă caracteristică a iteratorilor de inserare este că operatorul de incrementare `++` nu are niciun efect (nu modifică iteratorul).

În *STL* sunt definite 3 categorii de iteratori de inserare:

Iterator	Efect
<code>back_insert_iterator</code>	Inserează valoarea la sfârșitul containerului Disponibil pentru clasele <code>vector</code> , <code>deque</code> , <code>list</code> și <code>string</code>
<code>front_insert_iterator</code>	Inserează valoarea la începutul containerului Disponibil pentru clasele <code>deque</code> și <code>list</code>
<code>insert_iterator</code>	Inserează valoarea într-o poziție specificată (utilizând funcția <code>insert</code> a containerului)

Iteratorul pentru flux

Un iterator pentru flux (`stream_iterator`) este un adaptor utilizat pentru citirea (`istream_iterator`) sau scrierea (`ostream_iterator`) într-un flux.

4.6. Functorii

Un *functor* (obiect funcție) este un obiect al unei clase care are supraîncărcat operatorul (), denumit operatorul *apel de funcție*.

Să notăm cu x o astfel de clasă și xob un obiect al acestei clase. În cazul în care operatorul () este supraîncărcat pentru clasa x , putem utiliza construcția $xob(\dots)$, în paranteze fiind specificată lista parametrilor operatorului (). Această construcție reprezintă un apel al operatorului () pentru obiectul xob . Lista parametrilor poate fi vidă (functor generator) sau poate conține un parametru (functor unar), doi parametri (functor binar) sau oricărăți parametri sunt necesari.

O categorie specială de functori sunt *predicalele*, acestea fiind functori (unari sau binari) care returnează o valoare de tip `bool`.

În C sau $C++$ nu este posibil să transmiți unei funcții ca parametru o altă funcție. Este însă posibil să fie transmis un pointer de funcție (pointer care conține adresa unei funcții). Functorii corespund pointerilor de funcții din limbajul C .

În *STL* sunt predefinite mai multe categorii de functori. Pentru a utiliza acești functori, trebuie să includeți fișierul antet `functional`:

```
#include <functional>
```

Functori aritmetici

Functorii aritmetici sunt binari și realizează operația aritmetică specificată prin numele lor asupra celor doi parametri, returnând rezultatul operației. Aceștia sunt: plus (pentru adunare cu operatorul +), minus (pentru scădere cu operatorul -), multiplies (pentru înmulțire cu operatorul *), divides (pentru împărțire cu /), modulus (pentru restul împărțirii cu operatorul %).

Există și un functor aritmetic unar denumit negate (schimbă semnul algebraic al argumentului său, utilizând operatorul unar -).

Functori de comparare

Functorii de comparare sunt binari și returnează un rezultat de tip `bool`: `equal_to` (pentru egalitate cu operatorul ==), `not_equal_to` (pentru neegalitate cu operatorul !=), `less` (mai mic), `greater` (mai mare), `less_equal` (mai mic sau egal), `greater_equal` (mai mare sau egal).

Functori pentru operații logice

În *STL* există predefiniți trei functori pentru operații logice: `logical_not` (functor unar pentru negația logică cu operatorul ! a parametrului său), `logical_and` (functor binar pentru conjuncția logică cu operatorul && a celor doi parametri) și `logical_or` (pentru disjuncția logică a celor doi parametri utilizând operatorul ||).

4.7. Clasele adaptor pentru functori

Ca și în cazul containerelor și iteratorilor, există adaptori pentru functori, care au ca scop creșterea flexibilității în utilizare a functorilor predefiniți.

De exemplu, există doi adaptori de legătură (*binder*) care „leagă” unul dintre parametrii unui functor cu doi parametri, fixându-l pe o anumită valoare. Cu ajutorul acestor adaptori, un functor binar poate fi utilizat ca functor unar. Cei doi adaptori de tip *binder* sunt `bind1st` și `bind2nd` („leagă” primul, respectiv al doilea parametru).

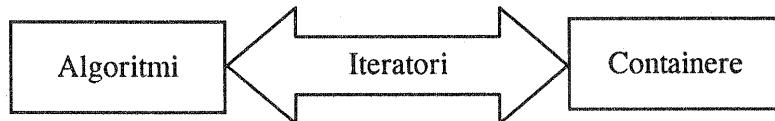
4.8. Algoritmii

Algoritmii sunt funcții generice (șablon), care nu aparțin claselor container și care implementează o serie de prelucrări fundamentale asupra elementelor containerelor, indiferent de tipul acestora.

Exceptând algoritmii numerici, pentru a utiliza algoritmii predefiniți în *STL*, trebuie să includeți fișierul antet `algorithm`:

```
#include <algorithm>
```

Algoritmii interacționează cu elementele containerelor prin intermediul iteratorilor :



Algoritmii au ca parametri iteratori și functori. De exemplu, de multe ori un algoritm prelucreză un anumit domeniu (*range*) de elemente ale unui container. Un astfel de domeniu este definit de doi iteratori (prim și ultim) și reprezintă toate elementele începând de la elementul indicat de iteratorul prim (inclusiv acesta), până la elementul indicat de iteratorul ultim (exclusiv acesta). Un astfel de domeniu este notat `[prim,ultim)`. De asemenea, când se specifică poziția de început a unui domeniu, aceasta se precizează prin intermediul unui iterator. Un alt exemplu frecvent întâlnit sunt algoritmii care verifică o anumită condiție pentru elementele containerului (condiția aceasta este transmisă ca parametru printr-un functor cu rezultat `bool`, adică un predicat).

În funcție de prelucrările realizate, algoritmii predefiniți din *STL* pot fi clasăți în 10 categorii. Unii dintre acești algoritmii sunt disponibili doar începând cu *C++ 11*. Pentru a indica acest lucru, menționăm ⁽¹¹⁾ după numele algoritmului.

Algoritmii care nu modifică elementele containerului

Acești algoritmi realizează în general verificări ale elementelor dintr-un domeniu, în funcție de o anumită condiție. Condiția poate fi ca elementele să fie egale sau ca un anumit predicat (transmis ca parametru) să aibă valoarea true.

Algoritm	Efect
all_of ⁽¹¹⁾	Verifică dacă toate elementele dintr-un domeniu verifică o anumită condiție
any_of ⁽¹¹⁾	Verifică dacă există un element într-un domeniu care verifică o anumită condiție
none_of ⁽¹¹⁾	Verifică dacă niciun element dintr-un domeniu nu verifică o anumită condiție
for_each	Aplică o anumită funcție pentru fiecare element dintr-un domeniu
find, find_if, find_if_not ⁽¹¹⁾ , find_end, find_first_of	Caută o valoare care îndeplinește o anumită condiție într-un domeniu
adjacent_find	Caută într-un domeniu două valori care îndeplinesc o anumită condiție, aflate pe poziții consecutive
count	Determină numărul de apariții ale unei valori într-un domeniu
count_if	Determină numărul de valori care îndeplinesc o anumită condiție dintr-un domeniu
mismatch	Parcurge în ordine elementele a două domenii și determină prima poziție pentru care elementele de pe poziția respectivă nu se „potrivesc” (nu îndeplinesc o anumită condiție)
equal	Parcurge în ordine elementele a două domenii și verifică dacă elementele corespondente din cele două domenii se „potrivesc” (îndeplinesc o anumită condiție)
is_permutation ⁽¹¹⁾	Verifică dacă un domeniu este o permutare a unui alt domeniu
search	Caută prima „potrivire” (în funcție de o anumită condiție) a unui subdomeniu într-un domeniu
search_n	Caută într-un domeniu prima apariție a unei succesiuni de n valori consecutive care îndeplinesc o anumită condiție.

Algoritmii care modifică elementele containerului

Acești algoritmi aplică o anumită prelucrare elementelor dintr-un domeniu (sau a primelor n elemente începând cu o poziție specificată – variantele al căror nume conține _n); rezultatul se obține în domeniul inițial sau se copiază într-un alt domeniu identificat doar prin poziția de început (variantele _copy).

Algoritm	Efect
copy copy_backward	Copiază (în ordine, respectiv în ordine inversă) elementele unui domeniu într-o poziție specificată
copy_n ⁽¹¹⁾	Copiază primele n elemente începând cu o poziție specificată la o altă poziție specificată

<code>copy_if⁽¹¹⁾</code>	Copiază elementele dintr-un domeniu care îndeplinesc o anumită condiție într-o poziție specificată
<code>move⁽¹¹⁾</code> <code>move_backward⁽¹¹⁾</code>	Mută (în ordine, respectiv în ordine inversă) elementele unui domeniu într-o poziție specificată
<code>swap</code> <code>iter_swap</code>	Interschimbă valorile a două obiecte. Pentru <code>swap</code> sunt transmise ca parametri referințe la obiectele care se interschimbă; pentru <code>iter_swap</code> sunt transmiși ca parametri iteratori care indică poziția elementelor care se interschimbă
<code>swap_ranges</code>	Interschimbă valorile dintr-un domeniu cu valorile unui alt domeniu, care începe la o poziție specificată
<code>transform</code>	Prelucrează elementele dintr-un domeniu (dacă prelucrarea este o funcție unară) sau din două domenii (dacă prelucrarea este o funcție binară) și stochează rezultatul prelucrării într-o poziție specificată
<code>replace</code> <code>replace_if</code>	Înlocuiește toate elementele dintr-un domeniu egale cu o anumită valoare, respectiv care îndeplinesc o anumită condiție cu o altă valoare
<code>replace_copy</code> <code>replace_copy_if</code>	Efect similar cu <code>replace</code> , respectiv <code>replace_if</code> , doar că rezultatul obținut este copiat începând cu o poziție specificată
<code>fill</code>	Atribuie o valoare tuturor elementelor dintr-un domeniu
<code>fill_n</code>	Începând cu o poziție specificată, se atribuie primelor n elemente o valoare specificată.
<code>generate</code>	Atribuie elementelor dintr-un domeniu valori generate succesiv de un generator (functor fără parametri)
<code>generate_n</code>	Începând cu o poziție specificată, se atribuie primelor n elemente valoare generate succesiv de un generator specificat ca parametru
<code>remove</code> <code>remove_if</code>	Șterge dintr-un domeniu toate elementele egale cu o anumită valoare, respectiv care îndeplinesc o anumită condiție
<code>remove_copy</code> <code>remove_copy_if</code>	Efect similar cu <code>remove</code> , respectiv <code>remove_if</code> , numai că rezultatul este copiat începând cu o anumită poziție
<code>unique</code> <code>unique_copy</code>	Dacă există într-un domeniu succesiuni de elemente egale, respectiv care îndeplinesc o anumită condiție, situate pe poziții consecutive, se elimină aceste succesiuni, păstrând doar primul element.
<code>reverse</code> <code>reverse_copy</code>	Inversează ordinea elementelor dintr-un domeniu. Rezultatul se obține în același domeniu, respectiv se copiază începând cu o poziție specificată
<code>rotate</code> <code>rotate_copy</code>	Realizează o rotație spre stânga cu un anumit număr de poziții a elementelor dintr-un domeniu
<code>random_shuffle</code>	Rearanjează în mod aleatoriu elementele unui domeniu
<code>shuffle⁽¹¹⁾</code>	Rearanjează în mod aleatoriu elementele unui domeniu utilizând un generator

Algoritmii de partitiorare

Acești algoritmi împart un domeniu în două subdomenii (primul subdomeniu este format din elementele pentru care predicatul transmis ca parametru returnează valoarea true, iar cel de-al doilea subdomeniu este format din elementele pentru care predicatul returnează valoarea false).

Algoritm	Efect
partition	Rearanjează elementele dintr-un domeniu astfel încât elementele pentru care predicatul specificat ca parametru returnează true sunt plasate înaintea elementelor pentru care predicatul returnează false; returnează ca rezultat un iterator care indică începutul celui de-al doilea subdomeniu
stable_partition	Efectul este similar cu partition, numai că se garantează că ordinea inițială a elementelor din fiecare domeniu se păstrează
partition_copy ⁽¹¹⁾	Copiază la o anumită poziție elementele dintr-un domeniu pentru care predicatul specificat ca parametru returnează true, iar la altă poziție elementele pentru care predicatul returnează false
partition_point ⁽¹¹⁾	Returnează un iterator care indică poziția primului element dintr-un domeniu specificat pentru care predicatul nu returnează true. Se consideră că domeniul a fost deja partitiorat
is_partitioned ⁽¹¹⁾	Returnează true dacă elementele unui domeniu sunt partitiorate în maniera descrisă la partition

Algoritmii de ordonare

Acești algoritmi ordonează sau verifică ordinea elementelor unui domeniu; pentru compararea a două elemente se utilizează operatorul = sau o funcție de comparare specificată ca parametru.

Algoritm	Efect
Sort	Ordonează elementele dintr-un domeniu
stable_sort	Ordonează elementele dintr-un domeniu, garantând că elementele echivalente își păstrează ordinea inițială
partial_sort	Realizează o ordonare parțială a elementelor dintr-un domeniu.
partial_sort_copy	Rezultatul se obține în același domeniu, respectiv se copiază într-un alt domeniu
is_sorted ⁽¹¹⁾	Returnează true dacă elementele dintr-un anumit domeniu sunt ordonate
is_sorted_until ⁽¹¹⁾	Returnează un iterator care indică poziția primului element care nu este în ordine într-un domeniu specificat
nth_element	Rearanjează elementele dintr-un domeniu astfel încât al n-lea element (considerând elementele sortate) să fie pe poziția sa corectă, elementele mai mici decât el fiind plasate înaintea sa

Algoritmii de căutare binară

Algoritmii din această categorie acționează asupra unui domeniu ordonat. și în acest caz pentru compararea a două elemente se utilizează operatorul implicit = sau o funcție de comparare specificată ca parametru.

Algoritm	Efect
binary_search	Căută binar o valoare într-un domeniu specificat și returnează <code>true</code> în cazul în care valoarea există în domeniu, respectiv <code>false</code> în caz contrar
lower_bound upper_bound	Returnează un iterator care indică poziția primului element din domeniul specificat care nu este mai mic, respectiv este mai mare decât o valoare specificată
equal_range	Returnează pentru un domeniu specificat, subdomeniul format numai din elemente egale cu o valoare specificată (subdomeniul fiind identificat printr-o pereche de iteratori: primul indicând poziția de început, al doilea poziția de sfârșit)

Algoritmii de lucru cu mulțimi

Acești algoritmi lucrează cu domenii ordonate. Domeniile obținute ca rezultat sunt de asemenea ordonate. Ordonarea elementelor se face considerând implicit operatorul < sau o funcție de comparare specificată ca parametru.

Algoritm	Efect
merge	Interclasează elementele unui domeniu cu elementele unui alt domeniu, copiind rezultatul începând cu o poziție specificată
inplace_merge	Interclasează domeniul [prim, mij] cu domeniul [mij, ultim), obținând rezultatul interclasării în domeniul [prim, ultim)
includes	Incluziune: verifică dacă toate elementele unui domeniu aparțin unui alt domeniu
set_union set_intersection set_difference set_symmetric_difference	Construiește la o poziție specificată un domeniu care este rezultatul reuniunii, intersecției, diferenței, respectiv diferenței simetrice a două domenii

Algoritmii de lucru cu heap-uri

În această categorie sunt inclusi algoritmii fundamentali de lucru cu *heap*-uri, algoritmi care vor fi apelați și de adaptorul `priority_queue`. Pentru compararea a două elemente se utilizează implicit operatorul < sau o funcție de comparare specificată ca parametru.

Algoritm	Efect
make_heap	Rearanjează elementele dintr-un domeniu astfel încât să constituie un <i>heap</i>
push_heap	Considerând că elementele din domeniul [prim, ultim-1) constituie un <i>heap</i> , se rearanjează elementele din domeniul [prim, ultim) astfel încât să constituie un <i>heap</i> (practic, se extinde un <i>heap</i> , prin inserarea elementului imediat următor)

<code>pop_heap</code>	Se extrage din <i>heap</i> -ul aflat în domeniul [prim, ultim) primul element și este interschimbat cu elementul de pe poziția indicată de <i>ultim</i> -1, apoi <i>heap</i> -ul este restructurat astfel încât elementele rămase în domeniul [prim, <i>ultim</i> -1) să constituie un <i>heap</i>
<code>sort_heap</code>	Sortează elementele dintr-un domeniu organizat ca un <i>heap</i>
<code>is_heap⁽¹¹⁾</code>	Returnează <code>true</code> dacă elementele dintr-un domeniu specificat constituie un <i>heap</i> , respectiv <code>false</code> în caz contrar
<code>is_heap_until⁽¹¹⁾</code>	Returnează un iterator care indică poziția primului element dintr-un domeniu specificat care încalcă proprietatea de <i>heap</i>

Algoritmii de maximum/minimum

Acești algoritmi determină cel mai mic/mare element dintr-un domeniu, returnând fie o referință, fie un iterator care indică poziția acestuia. Pentru compararea a două elemente se utilizează implicit operatorul < sau o funcție de comparare specificată ca parametru.

Algoritm	Efect
<code>min</code>	Returnează o referință către cea mai mică, respectiv cea mai mare valoare dintr-un domeniu specificat
<code>max</code>	
<code>minmax⁽¹¹⁾</code>	Returnează o pereche (pair) în care primul element este minimul, iar al doilea element este maximul dintr-un domeniu
<code>min_element</code>	Returnează un iterator care indică poziția celui mai mic/mare element dintr-un domeniu specificat
<code>max_element</code>	
<code>minmax_element⁽¹¹⁾</code>	Returnează o pereche de iteratori (pair) în care primul element indică minimul, iar al doilea element indică maximul

Algoritmii referitor la ordonarea lexicografică

Spunem că o secvență $x = (x_1, x_2, \dots, x_n)$ precedă (este mai mică) din punct de vedere lexicografic o secvență $y = (y_1, y_2, \dots, y_m)$ dacă există un indice k astfel încât $x_i = y_i$, pentru orice $i=1, k-1$ și fie $x_k < y_k$, fie $k > n$. Pentru compararea a două elemente, se utilizează implicit operatorul < sau o funcție de comparare specificată ca parametru.

Algoritm	Efect
<code>lexicographical_compare</code>	Funcția returnează un rezultat de tip <code>bool</code> care indică dacă primul domeniu specificat ca parametru precedă lexicografic al doilea domeniu specificat ca parametru
<code>next_permutation</code>	Funcția rearanjează elementele unui domeniu astfel încât să reprezinte permutarea imediat următoare/precedentă, în ordine lexicografică. Funcția returnează <code>true</code> dacă rearanjarea a fost posibilă, respectiv <code>false</code> în caz contrar
<code>prev_permutation</code>	

Algoritmii numerici

Algoritmii numerici realizează prelucrări ale unor secvențe numerice. Pentru a utiliza această categorie de algoritmi trebuie să includem fișierul antet numeric.

Algoritm	Efect
accumulate	Implicit, însumează elementele dintr-un anumit domeniu. Dacă este specificată, execută o altă operație în loc de adunare cu elementele din domeniu
adjacent_difference	Implicit, determină sirul diferențelor dintre elementele consecutive ale unui domeniu. Dacă este specificată, execută o altă operație în loc de scădere cu elementele din domeniu
inner_product	Implicit, determină suma produselor dintre elementele corespondente a două domenii. Este posibilă specificarea altor operații în loc de sumă și produs
partial_sum	Implicit, calculează sirul sumelor parțiale ale elementelor unui domeniu. Este posibilă specificarea unei alte operații în loc de adunare
iota ⁽¹¹⁾	Construiește o secvență crescătoare de valori consecutive, începând cu o valoare specificată

4.9. Clasa şablon pair

Deoarece apare frecvent necesitatea de a lucra cu o pereche de valori, a fost predefinită în biblioteca *STL* clasa şablon pair. Pentru a utiliza clasa pair trebuie să includem fișierul antet utility.

Declarația clasei pair

```
template <class T1, class T2> struct pair;
```

Observație

În declarație am utilizat cuvântul-cheie *struct*. În C++ tipul *struct* este similar cu tipul *class*, diferența constând în faptul că toți membrii tipului *struct* sunt publici.

Definiția clasei pair

```
template <class T1, class T2> struct pair
{//date membre
    T1 first;
    T2 second;
//constructor implicit
pair(): first(T1()), second(T2()) {}
//constructor cu doi parametri
pair(const T1& a, const T2& b): first(a), second(b) {}
//constructor de copiere cu conversie implicită
template<class U, class V> pair(const pair<U,V>& p):
    first(p.first), second(p.second) {}
};
```

Observații

1. Clasa `pair` conține două date membre publice: `first` și `second`, având tipul `T1`, respectiv `T2`.
2. Clasa `pair` are doi constructori (constructorul implicit și constructorul de inițializare care construiește o pereche pe baza a două valori specificate ca parametru) și un constructor de copiere.
3. Deși nu sunt explicit definiți în cadrul clasei, compilatorul generează automat destructorul și supraîncarcă operatorul de atribuire.
4. Începând cu `C++ 11`, în clasa `pair` a fost inclusă funcția membră `swap`, care interschimbă `first` cu `second`.

Funcția make_pair()

Funcția săalon `make_pair` permite construirea unei perechi pe baza a două valori specificate ca parametru, astfel:

```
template <class T1, class T2>
pair<T1, T2> make_pair (T1 x, T2 y)
{ return ( pair<T1, T2>(x,y) ); }
```

Operatorii de egalitate și operatorii relaționali

Operatorii de egalitate și operatorii relaționali sunt supraîncărcați pentru a funcționa cu perechi. Vom detalia doar operatorul `==` și operatorul `<`, deoarece semnificația celorlalți operatori poate fi descrisă cu ajutorul acestora.

Operatorul de egalitate ==

```
template <class T1, class T2>
bool operator==(const pair<T1, T2>& a, const pair<T1, T2>& b);
```

Perechea `a` este egală cu perechea `b` (`a==b`) dacă `a.first==b.first` și `a.second==b.second`.

Operatorul relațional <

```
template <class T1, class T2>
bool operator<(const pair<T1, T2>& a, const pair<T1, T2>& b);
```

Compararea a două perechi se realizează din punct de vedere lexicografic. Mai exact, perechea `a` este mai mică decât perechea `b` (`a<b`) dacă `a.first< b.first` sau `a.first==b.first` și `a.second<b.second`.

Exemplu

În exemplul următor vom demonstra modul de utilizare a constructorilor, a funcției `make_pair()` și a operatorilor relaționali, folosind perechi în care primul element (`first`) este de tip `string`, iar al doilea (`second`) este de tip `float`. Perechea `p1` este creată utilizând constructorul de inițializare, iar perechea `p2` este creată utilizând constructorul implicit, apoi este inițializată apelând funcția `make_pair()`.

```
#include <iostream>
#include <utility>
using namespace std;
int main()
{pair<string, float> p1("Ionescu Dan", 9.75);
 pair<string, float> p2;
 p2=make_pair("Popescu Ana", 9.43);
 cout<<p1.first<<' '<<p1.second<<'\n';
 cout<<p2.first<<' '<<p2.second<<'\n';
 if (p1==p2) cout<<"Perechi egale\n";
 else cout<<"Perechi neegale\n";
 if (p1<p2) cout<<"p1<p2";
 else cout<<"p1>=p2";
 p1=make_pair("Popescu Ana", 9.98);
 if (p1<p2) cout<<" Dupa schimbare: p1<p2\n";
 else cout<<" Dupa schimbare: p1>=p2\n";
 return 0;
}
```

După execuția acestui program, pe ecran se va afișa :

```
Ionescu Dan 9.75
Popescu Ana 9.43
Perechi neegale
p1<p2 Dupa schimbare: p1>=p2
```

4.10. Alocatorii

Alocatorii sunt clase care definesc modul de alocare a memoriei pentru containere din *STL*.

4.11. Aplicații

În capitolele următoare vom descrie diferite clase container și vom face aplicații cu acestea. Aici vom demonstra modul de funcționare a unor algoritmi din *STL* utilizând un tablou unidimensional (vector din C). În loc de iteratori vom utiliza pointeri. Acest lucru este posibil deoarece se realizează o conversie implicită a pointerilor în iteratori. Pentru exemplificare, luăm în considerare următoarele declarații :

```
int a[]={10,11,12,13,14,15,16,17,18,19};
int n=10;
```

Inversarea elementelor într-un tablou unidimensional

Declarația funcției `reverse()` :

```
template <class BidirectionalIterator>
void reverse (BidirectionalIterator prim,
              BidirectionalIterator ultim);
```

Funcția `reverse()` inversează ordinea elementelor din domeniul [prim, ultim]. Pentru a inversa întregul vector, putem apela funcția `reverse()` astfel:

```
| reverse(a, a+n);
```

După inversare, vectorul a este: {19, 18, 17, 16, 15, 14, 13, 12, 11, 10}.

Pentru a inversa doar elementele din vector de la poziția `st` (inclusiv) până la poziția `dr` (exclusiv), putem apela funcția `reverse()` astfel:

```
| reverse (a+st, a+dr);
```

Dacă `st=3` și `dr=7`, după inversare vectorul a este:

{10, 11, 12, 16, 15, 14, 13, 17, 18, 19}.

Eliminarea elementelor care îndeplinesc o anumită condiție

Declarația funcției `remove_if()`:

```
template <class ForwardIterator, class UnaryPredicate>
ForwardIterator remove_if (ForwardIterator prim,
                           ForwardIterator ultim,
                           UnaryPredicate pred);
```

Funcția `remove_if()` elimină din domeniul [prim,ultim] toate elementele pentru care predicatul returnează valoarea `true`, păstrând ordinea elementelor neeliminate. Funcția returnează un iterator care indică noua poziție de sfârșit a domeniului obținut după eliminare.

Funcția parurge domeniul și, la fiecare poziție pe care se află un element care îndeplinește condiția, caută primul element următor care nu îndeplinește condiția și îl plasează pe poziția curentă. Astfel, ordinea elementelor se păstrează, iar algoritmul de eliminare este liniar.

Să presupunem că dorim să eliminăm din vectorul a elementele pare. Vom construi o funcție cu rezultat `bool` care să verifice condiția dorită (această funcție va fi utilizată ca predicat).

```
| bool EstePar(int x) {return x%2==0;}
```

Pentru a elimina elementele pare din întregul vector a, putem apela funcția `remove_if()` astfel:

```
| int * psf=remove_if(a, a+n, EstePar);
```

Evident, funcția `remove_if()` nu modifică dimensiunea vectorului a. Prin urmare, vom actualiza după apel valoarea variabilei n (numărul de elemente din vectorul a), scăzând din pointerul `psf` în care am reținut noua poziție de sfârșit a domeniului, pointerul a care indică începutul acestuia:

```
| n=psf-a;
```

Dacă dorim să realizăm eliminarea elementelor pare doar de pe un interval din vectorul a, să spunem intervalul [a+st, a+dr], atunci trebuie să avem grijă să mutăm la stânga cu nr poziții toate elementele din intervalul [a+dr, a+n] (unde cu nr am notat numărul de elemente eliminate).

```

int st=3, dr=6, nr;
int * psf=remove_if(a+st,a+dr, EstePar);
nr=a+dr-psf;           //determin numarul de elemente eliminate
copy(a+dr,a+n,psf); //copiez restul vectorului
n-=nr;                 //actualizez dimensiunea vectorului

```

Transformarea sir

Ne propunem să aplicăm o prelucrare tuturor elementelor unui sir de caractere (în cazul nostru, dorim să transformăm toate literele mici din sir în majuscule). Pentru aceasta am putea utiliza algoritmul `for_each()` din *STL*.

```

template <class InputIterator, class Function>
Function for_each (InputIterator prim, InputIterator ultim,
                   Function fn);

```

Algoritmul `foreach()` aplică tuturor elementelor din domeniul [prim, ultim) o prelucrare descrisă de funcția fn.

```

#include <iostream>
#include <algorithm>
using namespace std;
string s;
void majuscula(char& c)
{ if (c>='a' && c<='z') c=c-'a'+'A'; }
int main()
{cin>>s;
 for_each(s.begin(), s.end(), majuscula);
 cout<<s;
 return 0; }

```

Observație

Funcției `for_each` i-am transmis ca parametri doi iteratori (`s.begin()` returnează un iterator care indică începutul sirul, iar `s.end()` returnează un iterator care indică poziția de după sfârșitul sirului) și numele funcției care realizează prelucrarea (`majuscula`).

Sortarea unui tablou unidimensional

Sortarea elementelor unui container se poate realiza cu ajutorul funcției `sort()`. Această funcție este supraîncărcată :

```

template <class RandomAccessIterator>
void sort(RandomAccessIterator prim, RandomAccessIterator ultim);
template <class RandomAccessIterator, class Compare>
void sort(RandomAccessIterator prim, RandomAccessIterator ultim,
          Compare comp);

```

Prima formă se poate utiliza pentru a sorta elementele din domeniul [prim, ultim] în ordine crescătoare (considerând pentru compararea elementelor operatorul <, care trebuie să fie definit pentru tipul elementelor containerului).

A doua formă permite specificarea funcției care se utilizează pentru compararea elementelor containerului (printron-un functor sau un pointer de funcție). Aceasta trebuie să fie o funcție cu doi parametri și trebuie să returneze un rezultat de tip bool. Rezultatul va fi true dacă primul parametru trebuie să fie plasat în vectorul sortat înaintea celui de-al doilea parametru.

Metoda de sortare utilizată este un algoritm creat de David Musser în 1997 denumit *introsort* (sau *introspective sort* – sortare introspectivă). Acest algoritm optimizează metoda de sortare *quicksort*, utilizând *heapsort*, pentru cazul în care adâncimea recursiei este prea mare. Complexitatea acestui algoritm este de $O(n \log n)$ în cazul cel mai defavorabil (unde cu n am notat numărul de elemente din domeniul ce trebuie sortat).

Sortarea crescătoare a unui tablou unidimensional de numere întregi

Pentru a sorta crescător întreg vectorul :

```
sort(a, a+n);
```

Pentru a sorta crescător doar domeniul [a+st, a+dr) :

```
sort(a+st, a+dr);
```

Sortarea crescătoare a unui tablou unidimensional de fracții

Revenim la clasa *fractie*, pe care am definit-o în capitolul al doilea. Pentru această clasă am supraîncărcat și operatorul <. Sortarea unui tablou unidimensional cu elemente de tip *fractie* se poate realiza astfel :

```
#include <iostream>
#include <algorithm>
#include "fractie.h"
using namespace std;
fractie a[1000];
int n;
void Citeste()
{cin>>n;
 for (int i=0; i<n; i++) cin>>a[i]; }
void Scrie()
{for (int i=0; i<n; i++) cout<<a[i]<<' ';
 cout<<'\n'; }

int main()
{Citeste();
 sort(a, a+n);
 Scrie();
 return 0; }
```

Concluzie

Dacă tipul elementelor containerului este un tip predefinit al limbajului sau un tip definit de utilizator care are supraîncărcat operatorul <, sortarea crescătoare a elementelor se realizează cu prima formă a funcției `sort()`.

Sortarea crescătoare a unui tablou unidimensional de elevi

Definim o structură denumită `elev` care va avea 3 câmpuri: numele elevului, prenumele elevului și media sa generală. Pentru acest tip de date, operatorul < nu este supraîncărcat, prin urmare vom defini o funcție de comparare a doi elevi (intenționăm să sortăm elevii lexicografic după nume, iar dacă există mai mulți elevi cu același nume, îi vom sorta lexicografic după prenume).

```
#include <fstream>
#include <algorithm>
using namespace std;
struct elev {string nume, pren; double mg; };
elev a[1000];
int n;
ifstream fin("elevi.in");
ofstream fout("elevi.out");
void Citeste()
{
    int i;
    fin>>n;
    for (i=0; i<n; i++) fin>>a[i].nume>>a[i].pren>>a[i].mg;
}
void Scrie()
{
    int i;
    for (i=0; i<n; i++)
        fout<<a[i].nume<<' '<<a[i].pren<<' '<<a[i].mg <<'\n';
}

bool ComparElevi (const elev& e1, const elev& e2)
{
    return e1.nume<e2.nume ||
           e1.nume==e2.nume && e1.pren<e2.pren;
}

int main()
{Citeste();
 sort(a,a+n, ComparElevi);
 Scrie();
 return 0;
}
```

În programul precedent am definit o funcție de comparare. Puteam defini `ComparElevi` ca functor, astfel:

```

struct ClasaComparElevi
{
    bool operator() (const elev el, const elev e2) const
    { return el.nume<e2.nume ||
        el.nume==e2.nume && el.pren<e2.pren; }
};

ClasaComparElevi ComparElevi;

```

Sortarea descrescătoare a unui tablou unidimensional

Pentru a sorta descrescător un tablou unidimensional, trebuie să utilizăm cea de-a doua formă a funcției de sortare, specificând ca al treilea parametru criteriul de sortare (un functor sau o funcție de comparare).

Pentru elemente de tipuri predefinite sau definite de utilizator cu operatorul **>** supraîncărcat, se poate utiliza functorul predefinit **greater<T>()**. Pentru a putea utiliza acest functor, trebuie să includem fișierul antet **functional**.

Functorul **greater** este definit astfel :

```

template<class T> struct greater:binary_function <T,T,bool>
{
    bool operator() (const T& x, const T& y) const
    { return x>y; }
};

```

Dacă a are componente de tip **int**, vom sorta descrescător elementele sale astfel :

```
| sort(a,a+n, greater<int>());
```

În același mod procedăm, de exemplu, și pentru un vector cu elemente de tip **fractie** (deoarece pentru această clasă am supraîncărcat operatorul **>**) :

```
| sort(a,a+n, greater<fractie>());
```

Sortarea descrescătoare a unui tablou unidimensional de elevi

În cazul în care operatorul **>** nu este supraîncărcat pentru tipul elementelor pe care trebuie să le sortăm, trebuie să definim un functor sau o funcție pentru compararea elementelor. În cazul vectorului de elevi, singura modificare pe care trebuie să o facem pentru a obține ordonare lexicografică descrescătoare este schimbarea criteriului de sortare în funcția **ComparElevi()** :

```

bool ComparElevi (const elev& el, const elev& e2)
{
    return el.nume>e2.nume ||
        el.nume==e2.nume && el.pren>e2.pren;
}

```

Sortarea stabilă

Sortarea stabilă conservă ordinea inițială a elementelor egale. Pentru sortare stabilă se utilizează funcția `stable_sort()`, de asemenea în două variante, asemănător cu funcția `sort()`.

Complexitatea sortării stabile este de $O(n \log n)$ (în cazul în care există memorie suplimentară suficientă) sau $O(n \log^2 n)$ în caz contrar.

Generarea de permutări

```
template <class BidirectionalIterator>
bool next_permutation (BidirectionalIterator prim,
                      BidirectionalIterator ultim);
template <class BidirectionalIterator, class Compare>
bool next_permutation (BidirectionalIterator prim,
                      BidirectionalIterator ultim, Compare comp);
```

Funcția `next_permutation()` rearanjează elementele din domeniul $[prim,ultim]$ astfel încât să constituie permutarea imediat următoare din punct de vedere lexicografic. Implicit, elementele se compară cu operatorul `<`. Dacă se utilizează cea de-a doua variantă, compararea elementelor va fi realizată cu ajutorul functorului `comp`. Funcția returnează `true` dacă rearanjarea a fost posibilă.

Vom utiliza această funcție pentru a genera în ordine lexicografică toate permutările de ordin n . În acest scop vom utiliza un vector `p`, pe care îl vom inițializa cu cea mai mică permutare din punct de vedere lexicografic ($1, 2, \dots, n$). La fiecare pas afișăm permutarea curentă și generăm permutarea următoare. Procedeul se repetă cât timp generarea următoarei permutări este posibilă.

```
#include <iostream>
#include <algorithm>
#define NMAX 50

using namespace std;
int n;
int p[NMAX];

int main()
{int i;
cout << "n="; cin>>n;
for (i=1; i<=n; i++) p[i]=i;
do
{for (i=1; i<=n; i++) cout<<p[i]<<' '; cout<<'\n';
while (next_permutation(p+1,p+n+1));
return 0;
}
```

Dacă dorim generarea permutărilor în ordine lexicografică inversă, vom inițializa vectorul p cu cea mai mare permutare ($n, n-1, \dots, 1$) și la fiecare pas, după afișarea permutării curente, generăm precedentă permutare apelând funcția `prev_permutation()`:

```
for (i=1; i<=n; i++) p[i]=n-i+1;
do
    {for (i=1; i<=n; i++) cout<<p[i]<<' '; cout<<'\n';}
while (prev_permutation(p+1,p+n+1));
```

Anagramele unui cuvânt

O anagramă este un cuvânt care are aceleași litere ca și cuvântul dat, eventual într-o altă ordine. De exemplu, cuvintele `tamara` și `armata` sunt anagrame.

Generarea anagramelor unui cuvânt este un procedeu similar cu generarea permutărilor. Există totuși un aspect la care trebuie să fim atenți: dacă începem generarea anagramelor cu cuvântul citit, vom obține doar anagramele mai mari din punct de vedere lexicografic decât acesta. Din acest motiv, înainte de a începe generarea, vom ordona crescător literele cuvântului citit (pentru a obține cea mai mică anagramă posibilă).

```
#include <iostream>
#include <algorithm>
#define NMAX 50

using namespace std;
string s;

int main()
{cin>>s;
 sort(s.begin(), s.end());
 do
    cout<<s<<'\n';
 while (next_permutation(s.begin(),s.end()));
 return 0;
}
```

Programul evidențiază faptul că funcția de generare a următoarei permutări funcționează și în cazul în care în domeniul specificat valorile nu sunt distincte.

Generarea sirului Fibonacci

Vom genera într-un tablou unidimensional primii n termeni din sirul Fibonacci, utilizând funcția `generate()`.

În acest scop, vom declara un functor `TF`, obiect al clasei `TermenF`. Clasa `TermenF` are datele membre `f1` și `f2` (doi termeni consecutivi din sirul Fibonacci) și operatorul `()` supraîncărcat. Acest operator returnează următorul termen din sirul Fibonacci și actualizează corespunzător `f1` și `f2`:

```

#include <iostream>
#include <algorithm>
int n;
int f[100];

class TermenF
{public:
    int f1, f2;
    int operator()() {int f3=f2+f1; f1=f2; f2=f3; return f3; }
} TF;

using namespace std;

int main()
{cout << "n="; cin>>n;
f[0]=TF.f1=0; f[1]=TF.f2=1; //initializare
generate(f+2, f+n, TF);      //generare
for (int i=0; i<n; i++) cout<<f[i]<<' ';
return 0;
}

```

4.12. Exerciții și probleme recapitulative

- Să considerăm un tablou unidimensional a cu elemente reale, plasate pe pozițiile 1, 2, ..., n. Care dintre următoarele variante sortează crescător elementele vectorului a ?
 - sort(1, n);
 - sort(a+1, a+n);
 - sort(a+1, a+n+1);
 - În acest caz nu este posibilă sortarea cu sort().
- Care dintre următoarele expresii are valoarea true dacă în containerul C nu există elemente (este vid) :
 - C.begin() == 0
 - C.end() == 0
 - C.begin() == C.end() + 1
 - C.begin() == C.end()
- Care dintre următoarele afirmații sunt adevărate ?
 - Orice container, indiferent de tipul acestuia, trebuie să disponă de constructori privați.
 - Algoritmii interacționează cu containerele prin intermediul iteratorilor.
 - Un obiect al unei clase şablon se numeşte functor.
 - Pentru a indica un element al unui container putem utiliza un iterator.
 - Orice container conține funcțiile membre begin() și end().
 - Orice iterator, indiferent de tipul acestuia, suportă operația de decrementare.
 - Un predicat este un functor care returnează o valoare de tip bool.
 - Operatorul + este supraîncărcat astfel încât să permită adunarea dintre un iterator bidirecțional și un întreg.
 - Orice iterator poate fi dereferențiat cu operatorul unar *.

- j. Algoritmii sunt funcții membre ale claselor container.
 - k. Pentru a sorta descrescător elementele unui container, înțotdeauna poate fi utilizat ca parametru al funcției `sort()` functorul predefinit `greater`.
4. Generați aleatoriu un tablou unidimensional cu n elemente naturale < 10000 , utilizând funcția `generate()`.
5. Să considerăm un tablou unidimensional cu numere naturale cu cel puțin trei cifre. Utilizând funcția `for_each()`, eliminați din fiecare element al tabloului cifra din mijloc (dacă elementul are număr impar de cifre), respectiv cifrele din mijloc (dacă elementul are număr par de cifre).

6. Să considerăm următoarea structură:

```
struct DataC {int an, luna, zi; };
```

Din fișierul text `date1.in` se citește un număr natural n , apoi o succesiune de n date calendaristice, câte una pe o linie, scrise sub forma:

`zi-luna-an`

Datele calendaristice vor fi memorate într-un vector cu elemente de tip `DataC`. Ordonați datele cronologic, folosind funcția `sort()` din *STL* în trei variante:

- supraîncărcați operatorul `<`;
- implementați criteriul de sortare ca funcție;
- implementați criteriul de sortare ca functor.

În mod similar, sortați și datele stocate (în același mod) în fișierul `date2.in`.

Utilizând algoritmii din *STL*, determinați și afișați:

- datele calendaristice care se află în ambele fișiere;
- datele calendaristice din cele două fișiere, în ordine cronologică.

7. Fișierul `dictionar.in` conține cel mult 10000 de linii, pe fiecare linie fiind scrisă o definiție de forma:

`termen=descriere`

Rezolvați, utilizând algoritmii din *STL*, următoarele cerințe:

- Citiți conținutul fișierului `dictionar.in` și construiți un tablou unidimensional ale cărui componente sunt perechi (pair) formate dintr-un termen și descrierea corespunzătoare.
- Sortați termenii din dicționar în ordine lexicografică.
- Citiți de la tastatură un cuvânt și căutați-l eficient în dicționar. Dacă ați găsit în dicționar cuvântul, afișați descrierea corespunzătoare. În caz contrar afișați mesajul `Nu există`.
- Căutați în dicționar și afișați pe ecran toate anagramele cuvântului citit de la tastatură.
- Știind că numele proprii încep cu majusculă, rearanjați termenii din dicționar astfel încât la început să fie numele proprii în ordine lexicografică, apoi restul cuvintelor, de asemenea în ordine lexicografică.

5. Containere secvențiale

Containerele secvențiale predefinite în *STL* sunt *vector*, *deque*, *list* și, începând cu *C++ 11*, *array* și *forward_list*.

În capitolul precedent am menționat o serie de elemente comune tuturor containerelor din *STL*. În continuare vom prezenta elementele specifice fiecărui tip de container.

5.1. Clasa *vector*

Pentru a utiliza această clasă, trebuie să includem fișierul antet *vector*:

```
#include <vector>
```

Clasa *vector* implementează o structură de date similară cu tablourile unidimensionale (vectorii din limbajul *C*). Elementele containerului sunt de același tip și sunt memorate într-o zonă continuă de memorie. Ca urmare, elementele containerului pot fi accesate atât prin intermediul iteratorilor, cât și prin intermediul pointerilor. Iteratorii containerului *vector* sunt iteratori cu acces aleatoriu.

Diferența semnificativă constă în faptul că memoria necesară pentru elementele vectorului este alocată dinamic. În acest scop, clasa *vector* utilizează un alocator (implicit, acesta este un obiect al clasei predefinite *allocator*).

Constructorii și destructorul

Constructorii clasei *vector* sunt:

- *Constructorul implicit* (fără parametri) – creează un vector vid (fără elemente).

Exemplu

```
vector<int> a;
```

Efect: creează un vector vid cu elemente de tip *int*

- *Constructor de umplere (fill)* – creează un vector cu număr specificat de elemente; în cazul în care se specifică și o valoare ca al doilea parametru, elementele vectorului sunt inițializate cu valoarea specificată.

Exemplu

```
| vector<int> a(100, 0);
```

Creează un vector cu 100 de elemente de tip int și inițializează cu 0 toate elementele.

```
| vector<double> b(1000);
```

Creează un vector cu 1000 de elemente de tip double, fără inițializare.

- *Constructor bazat pe un domeniu (range)* – creează un vector în care copiază în ordine toate elementele dintr-un domeniu specificat.

Exemplu

```
| string s;
cin>>s;
vector<char> d(s.begin(), s.end());
```

Creează un vector de dimensiune egală cu dimensiunea sirului s și copiază în vector fiecare caracter din sirul s.

- *Constructor de copiere* – creează o copie a unui vector.
- *Destructorul* – distrugе obiectul, eliberând memoria alocată dinamic.

Complexitate

Pentru constructorul de umplere cu inițializare, constructorul bazat pe un domeniu și constructorul de copiere, complexitatea este liniară; ceilalți constructori necesită timp constant.

Funcțiile membre care returnează iteratori

Clasa vector conține funcțiile begin(), end(), rbegin(), rend() cu semnificația uzuală. Începând cu C++ 11, există și funcțiile corespondente care returnează const_iterator (cbegin(), cend(), crbegin(), crend()).

Funcțiile membre referitoare la dimensiunea vectorului

Alături de funcțiile uzuale size(), empty() și max_size() în clasa vector sunt definite și următoarele funcții membre publice:

- capacity()

```
| size_type capacity() const noexcept;
```

Returnează dimensiunea zonei de memorie alocată pentru vector, exprimată în număr de elemente; memoria alocată nu este obligatoriu egală cu numărul efectiv de elemente din vector (ar putea fi mai mare, pentru a permite inserarea unor noi elemente fără a fi necesară reallocarea memoriei).

- `reserve()`

```
void reserve (size_type n);
```

Asigură faptul că vectorul are memorie alocată pentru cel puțin n elemente. Dacă memoria alocată pentru vector este mai mică de n elemente, se alocă memoria necesară (posibil în timp liniar), în caz contrar apelul funcției neavând niciun efect.

- `resize()`

```
void resize (size_type n);  
void resize (size_type n, const value_type& val);
```

Redimensionează spațiul de memorie alocat vectorului astfel încât să conțină exact n elemente. Dacă dimensiunea curentă a vectorului este mai mare de n elemente, se păstrează doar primele n elemente ale vectorului (celelalte fiind distruse). Dacă dimensiunea curentă a vectorului este mai mică de n elemente, se adaugă elementele necesare la sfârșitul vectorului (eventual inițializându-se elementele adăugate cu valoarea `val`).

- `shrink_to_fit()(11)`

```
void shrink_to_fit();
```

Redimensionează memoria alocată vectorului astfel încât să fie egală cu numărul de elemente din vector.

Accesori

Accesul la elementele unui vector se realizează cu ajutorul următoarelor funcții membre publice :

- operatorul de indexare `[]`

```
reference operator[] (size_type n);  
const_reference operator[] (size_type n) const;
```

Returnează o referință către elementul situat pe poziția n în vector (pozițiile fiind numerotate de la 0).

- `at()`

```
reference at (size_type n);  
const_reference at (size_type n) const;
```

Returnează o referință către elementul situat pe poziția n în vector. Spre deosebire de operatorul de indexare, funcția `at()` verifică validitatea pozitiei n și „aruncă” excepția `out_of_range` în cazul în care $n \geq size()$.

- `front()`

```
reference front();  
const_reference front() const;
```

Returnează o referință către primul element din vector (spre deosebire de `begin()`, care returnează un iterator).

- `back()`

```
reference back();
const_reference back() const;
```

Returnează o referință către ultimul element al vectorului (spre deosebire de `end()`, care returnează un iterator ce indică poziția de după ultimul element).

- `data()`⁽¹¹⁾

```
value_type* data() noexcept;
const value_type* data() const noexcept;
```

Returnează un pointer care conține adresa de început a zonei de memorie în care sunt stocate elementele vectorului.

Toate funcțiile de acces se execută în timp constant.

Funcțiile membre care modifică vectorul

Ca pentru orice container, este supraîncărcat operatorul de atribuire `=`, funcția membră `swap()`, funcția `clear()`, funcțiile `insert()` și `erase()`. În plus, sunt definite următoarele funcții membre publice care modifică vectorul :

- `push_back()`

```
void push_back (const value_type& val);
```

Inserează un nou element la sfârșitul vectorului și copiază în acest element valoarea `val`. Dimensiunea vectorului va crește cu 1 și, ca urmare, este posibil să se execute reallocarea memoriei.

- `pop_back()`

```
void pop_back();
```

Elimină și distrugе ultimul element din vector. Dimensiunea vectorului va scădea cu 1.

- `insert()`

```
iterator insert(const_iterator poz, const value_type& val);
iterator insert(const_iterator poz, size_type n,
                const value_type& val);
template <class InputIterator>
iterator insert(const_iterator poz, InputIterator prim,
                InputIterator ultim);
```

Se inserează unul sau mai multe elemente în poziția specificată de iteratorul `poz`. Ca urmare, dimensiunea vectorului va crește cu numărul de elemente inserate și este posibil să se execute reallocarea memoriei. Prima variantă inserează valoarea `val`, a doua variantă (*fill*) inserează `n` elemente având valoarea `val`, a treia

variantă inserează elementele din domeniul [prim, ultim]. Funcția returnează un iterator care indică primul element inserat.

- `erase()`

```
| iterator erase (iterator poz);
| iterator erase (iterator prim, iterator ultim);
```

Elimină și distrugе din vector elementul indicat de iteratorul poz, respectiv toate elementele din domeniul [prim, ultim]. Ca urmare, dimensiunea vectorului va scădea cu numărul de elemente eliminate. Funcția returnează un iterator care indică poziția de după ultimul element eliminat.

- `assign()`

```
| template <class InputIterator>
| void assign (InputIterator prim, InputIterator ultim);
| void assign (size_type n, const value_type& val);
```

Funcția realizează o atribuire, înlocuind elementele vectorului cu elementele din domeniul [prim, ultim], respectiv cu n elemente cu valoarea val. În cazul în care capacitatea vectorului este mai mică decât numărul de elemente din domeniu, respectiv decât n, se alocă memoria necesară.

Complexitatea operațiilor de inserare/eliminare

<code>push_back()</code>	<code>pop_back()</code>	<code>insert()</code>	<code>erase()</code>	<code>assign()</code>
O(1)	O(1)	O(n)	O(n)	O(n)

Observații

1. Funcțiile `pop_back()`, `erase()` și `clear()` nu eliberează memoria alocată elementelor eliminate. Nici funcțiile `resize()` sau `reserve()` nu reduc dimensiunea memoriei alocate. Pentru ca aceasta să fie exact memoria necesară, se poate utiliza începând cu C++ 11 algoritmul `shrink_to_fit()`. Pentru compilatoare mai vechi, putem apela la o „șmecherie”: creez un vector temporar pe care îl interschimb cu vectorul pe care îl redimensionăm. Crearea vectorului temporar o vom realiza utilizând constructorul de copiere. Să presupunem că vectorul este denumit v, iară o modalitate de a dimisiona exact memoria alocată vectorului v în timp liniar:

```
| vector<T>(v).swap(v);
```

2. Începând cu C++ 11, funcțiile `insert()` și `assign()` au o câte o variantă care primește ca parametru o listă de inițializare. O listă de inițializare este un obiect al clasei `initializer_list` care se construiește automat pe baza unei liste de valori (o succesiune de valori separate prin virgulă, încadrată între acolade). De asemenea, au fost introduse încă două funcții membre denumite `emplace()` și `emplace_back()`, care construiesc și inserează un element într-o anumită poziție.

3. Deși nu le-am prezentat, începând cu C++ 11 există versiuni ale unor funcții membre care au parametri de tip `value_type&&`:

```
void push_back (value_type&& val);
iterator insert (const_iterator poz, value_type&& val);
```

Construcția `Tip&&` a fost introdusă începând cu C++ 11 pentru a indica o referință la o valoare de tip *rvalue* (*right-value*, valoare care poate fi plasată doar în membrul drept al unei atribuiri). În limbajul C, o valoare temporară care putea fi plasată doar în membrul drept al unei atribuiri nu era modificabilă. Acest nou tip indică faptul că o valoare temporară care este plasată în membrul drept al unei atribuiri, poate fi modificată după ce a fost creată. Acest tip a fost introdus din motive de eficiență a implementării, pentru a indica mutarea valorii respective¹.

De exemplu, în variantele de mai sus, valoarea specificată ca parametru al funcției `push_back()`, respectiv `insert()` este mutată în vector (nu copiată).

Operatorii relaționali

Pentru clasa `vector` sunt supraîncărcați toți operatorii relaționali, compararea a doi vectori realizându-se din punct de vedere lexicografic.

Aplicația 1. Calorii

A devenit o adevărată artă să alcătuiești un meniu care să fie bogat dar să conțină cât mai puține calorii. Fiecare produs are înscris pe el procentul de grăsimi pe care le conține și numărul de calorii. Din păcate, acest lucru nu se întâmplă și la piață. Văzând aceasta, un țăran inventiv, care oferă spre vânzare un număr de n produse, a făcut un mic calcul și a determinat, pentru fiecare dintre produsele oferite spre vânzare, numărul de calorii pe care le conține întreaga cantitate din produsul respectiv. Țăranul a scris pe câte o etichetă numerele astfel determinate și a pus fiecare etichetă lângă produsul corespunzător, având grija ca după fiecare vânzare să corecteze în mod corespunzător numărul înscris pe etichetă. O gospodină ce dispune de o anumită sumă de bani S pentru cumpărături dorește să cumpere produse de la acel țăran astfel încât să cheltuiască întreaga sumă S de care dispune, dar să ducă acasă cât mai multe calorii. Gospodinei ii este indiferent ce produse cumpără și în ce cantități, scopul fiind ca produsele cumpărate să conțină în totalitate cel mai mare număr de calorii.

Cerință

Să se determine cantitatea maximă de calorii ce poate fi cumpărată.

Date de intrare

Fișierul de intrare `calorii.in` va conține pe prima linie un număr natural n, reprezentând numărul de produse. Fiecare dintre următoarele n linii va conține câte

1. Explicația este extrem de succintă. Thomas Becker a realizat o prezentare foarte clară și detaliată care poate fi consultată la adresa : http://thbecker.net/articles/rvalue_references/section_01.html.

două valori naturale, reprezentând numărul de calorii determinat de țăran pentru produs și cât costă întreaga cantitate din produsul respectiv. Ultima linie a fișierului de intrare va conține o valoare naturală și reprezentând suma de care dispune gospodina pentru cumpărături.

Date de ieșire

Fișierul de ieșire `calorii.out` va conține pe prima linie o valoare reală cu patru zecimale, reprezentând numărul maxim de calorii conținute în produsele cumpărate.

Restricții

- $1 \leq n \leq 10000$
- $1 \leq s \leq 2000000000$
- Numărul de calorii și prețurile sunt valori naturale ≤ 30000 .
- Rezultatul afișat este considerat corect dacă eroarea este mai mică decât 10^{-3} .

Exemple

<code>calorii.in</code>	<code>calorii.out</code>	Explicații
5	1585.7143	Rezultatul se obține prin cumpărarea integrală a produselor 2, 4 și 5 și a 57.1428571% din produsul 3. Prin calcul se observă faptul că:
100 5		$300+700+500+57.1428571*150/100 =$
300 4		1585.71428565
150 7		
700 2		
500 20		
30		

(Olimpiada Municipală de Informatică, Iași, 2010)

Soluție

Problema este o reformulare a problemei rucsacului, varianta continuă. Metoda de rezolvare este *Greedy*:

- sortăm produsele după numărul de calorii pe care le pot obține din produsul respectiv cu 1 leu;
- cât timp produsul curent poate fi cumpărat în întregime (prețul său este mai mic sau egal decât suma disponibilă), îl cumpărăm (adăugăm la total caloriile achiziționate, iar din suma disponibilă scădem prețul produsului);
- la final, dacă mai există un produs, cumpărăm un procent din acesta, în funcție de suma disponibilă ; caloriile obținute se calculează cu regula de trei simplă.

Pentru implementare, definim clasa `Produs`, care are două date membre : `cal` (numărul de calorii pentru întregul produs) și `pret` (prețul produsului) ; supraîncărcăm ca funcție membră operatorul `>` :

```
struct Produs
{ int cal, pret;
  bool operator>(const Produs & p) const
  { return cal*p.pret>pret*p.cal; }
};
```

Declarăm vectorul de produse p astfel :

```
| vector<Produs> p;
```

Pentru a nu fi necesare reallocări de memorie, rezervăm de la început memoria necesară pentru vectorul p :

```
| p.reserve(n);
```

Din fișierul de intrare citim succesiv câte un produs (în variabila auxiliară x de tip Produs) și adăugăm în vector produsul x apelând funcția `push_back()` :

```
| Produs x;
for (int i=0; i<n; i++)
    {fin>>x.cal>>x.pret;
     p.push_back(x); }
```

Pentru sortarea descrescătoare a produselor din vectorul p, apelăm funcția `sort()` :

```
| sort(p.begin(), p.end(), greater<Produs>());
```

Pentru a rezolva problema, parcurgem elementele vectorului p folosind un iterator it, atât timp cât produsul curent poate fi cumpărat integral :

```
| vector<Produs>::iterator it;
for (it=p.begin(); it!=p.end(); it++)
    if (it->pret <= S)
        { total+=it->cal;
          S-=it->pret; }
    else break;
```

Aceeași parcurgere o puteam scrie „normal”, utilizând operatorul de indexare :

```
| for (int j=0; j<n; j++)
    if (p[j].pret <= S)
        { total+=p[j].cal; S-=p[j].pret; }
    else break;
```

La final, dacă mai există un produs disponibil, cumpărăm o fracțiune din acest produs, în funcție de suma rămasă :

```
| if (it!=p.end()) total+=(double)S*it->cal/it->pret;
```

Programul complet arată astfel :

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
#include <iomanip>
using namespace std;
ifstream fin("calorii.in");
ofstream fout("calorii.out");
struct Produs
```

```

{int cal, pret;
bool operator>(const Produs & p) const
{ return cal*p.pret>pret*p.cal; }
};

int main()
{int n, S;
double total=0;
Produs x;
fin>>n;
vector<Produs> p;
p.reserve(n);
for (int i=0; i<n; i++)
{fin>>x.cal>>x.pret;
p.push_back(x); }
fin>>S;

sort(p.begin(), p.end(), greater<Produs>());

vector<Produs>::iterator it;
for (it=p.begin(); it!=p.end(); it++)
if (it->pret<=S)
{ total+=it->cal; S-=it->pret; }
else break;
if (it!=p.end()) total+=(double)S*it->cal/it->pret;

fout<<setprecision(12)<<total<<'\n'; fout.close();
return 0;
}

```

Aplicația 2. Zid

Ionel a construit n turnuri formate din h_1, h_2, \dots , respectiv h_n cuburi (fiecare cub având latura 1). Tatăl său i-a propus să facă un zid de lungime n . În acest scop, trebuie să fie alese niște turnuri dintre cele n construite astfel încât, rearanjând cuburile din turnurile alese, să se obțină un zid de lungime n . Zidul trebuie să fie format din exact n turnuri având aceeași înălțime.

Date de intrare

Fișierul de intrare *zid.in* conține pe prima linie un număr natural n care reprezintă numărul de turnuri. Pe cea de-a doua linie sunt scrise n numere naturale separate prin câte un spațiu, reprezentând înălțimile celor n turnuri.

Date de ieșire

Fișierul de ieșire *zid.out* va conține pe prima linie un număr natural k , reprezentând numărul de turnuri alese pentru a construi zidul. Pe cea de-a doua linie vor fi

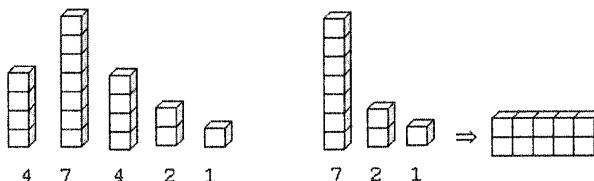
scrise k numere naturale separate prin câte un spațiu, reprezentând înălțimile celor k turnuri alese pentru construirea zidului.

Restricții și precizări

- $0 < n \leq 1000$
- $0 < h_i \leq 1000$, pentru orice $i=1, 2, \dots, n$

Exemplu

zid.in	zid.out	Explicație
5 4 7 4 2 1	3 7 2 1	Se formează un zid de lungime n și înălțime 2



(.campion 2005)

Soluție

Problema este o aplicație directă a principiului cutiei al lui *Dirichlet*. Pentru a alege o submulțime de turnuri cu suma înălțimilor divizibilă cu n, vom construi sirul sumelor parțiale $s_1=h_1$; $s_2=h_1+h_2$; ... $s_n=h_1+h_2+\dots+h_n$. Există două cazuri:

1. În sirul sumelor parțiale există $1 \leq i \leq n$, astfel încât $s_i \% n == 0$. În acest caz, o soluție posibilă este formată din turnurile 1, 2, ..., i.
2. În sirul sumelor parțiale $s_i \% n \neq 0$, pentru orice $1 \leq i \leq n$. În acest caz avem n resturi care iau valori în mulțimea $\{1, 2, \dots, n-1\}$. Conform principiului cutiei al lui *Dirichlet*, există $1 \leq i < j \leq n$, astfel încât $s_i \% n == s_j \% n$. O soluție posibilă este i+1, i+2, ..., j (deoarece $(s_j - s_i) \% n == 0$).

Există numeroase modalități de a implementa acest algoritm. Scopul nostru este de a ilustra modul de funcționare a componentelor *STL*, prin urmare:

1. Citim succesiv cele n înălțimi și le adăugăm în vectorul h (pentru care am rezervat în prealabil memoria necesară):

```
vector<int> h;
h.reserve(n);
for (i=0; i<n; i++)
    { fin>>x; h.push_back(x); }
```

2. Construim sirul sumelor parțiale s, utilizând algoritmul numeric *partial_sum()*:

```
vector<int> s;
s.reserve(n);
partial_sum(h.begin(), h.end(), s.begin());
```

3. Transformăm sumele parțiale în resturi, utilizând algoritmul `for_each()`:

```
| for_each(s.begin(), s.begin() + n, rest);
```

Functia `rest()` transformă un număr întreg în restul împărțirii sale la n :

```
| void rest (int& x) { x = x % n; }
```

4. Căutăm, utilizând algoritmul `find()`, o valoare egală cu 0 în sirul resturilor, reținând în iteratorul `poz` poziția pe care se află prima valoare egală cu 0:

```
| vector<int>::iterator poz;  
| poz = find(s.begin(), s.begin() + n, 0);
```

Dacă `poz != s.begin() + n`, ne aflăm în primul caz (a fost identificat un rest nul), în caz contrar, ne aflăm în cel de-al doilea caz și trebuie să identificăm două resturi egale. Programul complet arată astfel:

```
#include <iostream>
#include <algorithm>
#include <numeric>
#include <vector>
using namespace std;
ifstream fin("zid.in");
ofstream fout("zid.out");
int n;

void rest (int& x) { x = x % n; }

int main()
{ int i, j, k, x, cate, gasit;
  fin >> n;
  vector<int> h;
  h.reserve(n);
  for (i = 0; i < n; i++)
    { fin >> x;
      h.push_back(x);
    }

  vector<int> s;
  s.reserve(n);
  partial_sum(h.begin(), h.end(), s.begin());

  for_each(s.begin(), s.begin() + n, rest);

  vector<int>::iterator poz;
  poz = find(s.begin(), s.begin() + n, 0);

  if (poz != s.begin() + n) //cazul 1
    { cate = poz - s.begin() + 1;
      fout << cate << '\n';
      for (i = 0; i < cate; i++) fout << h[i] << ' ';
      fout << '\n';
    }
}
```

```

else //cazul 2
{for (gasit=i=0; i<n && !gasit; i++)
    for (j=i+1; j<n && !gasit; j++)
        if (s[i]==s[j]) gasit=1;
j--; i--;
fout<<j-i<<'\\n';
for (k=i+1; k<=j; k++) fout<<h[k]<<' ' ; fout<<'\\n';
}
fout.close();
return 0;
}

```

Aplicația 3. Lucrul cu matrice

În *STL* nu există o clasă specială pentru lucrul cu matrice. O matrice poate fi implementată ca un vector de vectori.

De exemplu, să implementăm o matrice *v* cu *n* linii și *m* coloane și componente de tip *int*. În acest scop vom declara un vector cu componente de tip *vector<int>* și alocăm memorie pentru vectorul de *n* linii :

```
| vector< vector<int> > v(n);
```

Vom „umple” matricea cu numerele naturale de la 1 la *n*m* (considerând parcurgerea pe linii). Pentru a nu realoca memorie la fiecare element adăugat, pentru fiecare linie alocăm memorie pentru *m* elemente :

```

int cont=0;
for (i=0; i<n; i++)
{v[i].reserve(m);
 for (j=0; j<m; j++) v[i].push_back(++cont);
}
```

În continuare putem prelucra această matrice ca pe o matrice „normală” (operatorul de indexare fiind supraîncărcat) :

```

#include <iostream>
#include <vector>
using namespace std;
int main()
{int n, m, i, j, x;
cin>>n>>m;
vector< vector<int> > v(n);
int cont=0;
for (i=0; i<n; i++)
{ v[i].reserve(m);
 for (j=0; j<m; j++) v[i].push_back(++cont); }
for (i=0; i<n; i++)
{ for (j=0; j<m; j++) cout<<v[i][j]<<' '; cout<<'\\n'; }
return 0;
}
```

Clasa specializată vector<bool>

În fișierul antet `vector` este definită și o clasă specializată: `vector<bool>` (vectori având elemente de tip `bool`). Spunem că această clasă este specializată deoarece, pentru economie de memorie, o valoare de tip `bool` este implementată pe un *bit*, nu pe un *byte*.

Pentru această clasă, iteratorii și pointerii au fost modificați astfel încât să indice un bit, iar tipul `reference` este definit ca o clasă ce permite accesarea fiecărui bit din reprezentare. În acest mod, utilizatorul acestui container specializat beneficiază de o interfață similară cu cea a unui `vector`, fără a conștientiza efectiv lucrul cu biți.

Functii membre suplimentare

- `flip()`

```
| void flip();
```

schimbă în timp liniar fiecare bit din container (`true` devine `false`, iar `false` devine `true`).

- `swap()`

```
| void swap (vector& x);
| static void swap (reference ref1, reference ref2);
```

Funcția `swap()` este supraîncărcată astfel încât să permită, prin cea de-a doua formă, interschimbarea a două elemente ale unui `vector<bool>`.

Clasa specializată hash<vector<bool>>

Începând cu C++ 11, este definită o specializare și pentru clasa `vector<bool>`, denumită `hash`. Această specializare are supraîncărcat operatorul apel de funcție `()`, care returnează o valoare `hash` calculată în funcție de toate elementele din vectorul respectiv. Această valoare `hash` permite containerelor `vector<bool>` să poată fi utilizate drept cheie în containere asociative nesortate.

Aplicația 4. Ciurul lui Eratostene

Vom genera toate numerele prime <`VMAX` prin metoda ciurului lui Eratostene. Ciurul va fi reprezentat ca un `vector<bool>`, inițializat cu valoarea `true` (`ciur[i]` va fi `true` dacă i este număr prim, respectiv `false` în caz contrar).

Parcurgem ciurul și, dacă i este număr prim, atunci eliminăm din ciur toți multiplii lui i .

```
| #include <iostream>
| #include <vector>
```

```
#define VMAX 100000
using namespace std;
vector<bool> ciur(VMAX, true);

int main()
{int i, j;
for (i=2; i*i<VMAX; i++)
    if (ciur[i]) //i este prim
        //elimin din ciur multiplii neeliminati ai lui i
        for (j=i; j*i<VMAX; j++) ciur[j*i]=false;
//afisez numerele prime
for (i=2; i<VMAX; i++)
    if (ciur[i]) cout<<i<<' ';
return 0;
}
```

Aplicația 5. Reprezentarea unui graf prin liste de adiacență

Vom reprezenta un graf neorientat prin liste de adiacență și apoi vom descompune graful în componente conexe, utilizând parcurgerea *DFS*.

Pentru a reprezenta listele de adiacență, utilizăm un vector *v* cu *NMAX* elemente de tip *vector<int>* (unde cu *NMAX* am notat numărul maxim de vârfuri din graf).

```
| vector< vector<int> > v(NMAX);
```

Citim succesiv câte o muchie *x* *y* din graf și inserăm pe *x* în lista de adiacență a lui *y* și pe *y* în lista de adiacență a lui *x*:

```
| fin>>x>>y;
v[x].push_back(y); v[y].push_back(x);
```

În cazul în care graful este orientat, *y* nu mai trebuie adăugat în lista de adiacență a lui *x*.

Pentru a reține dacă un vârf a fost sau nu vizitat, utilizăm un *vector<bool>*, denumit *viz*, pe care îl inițializăm cu *false*.

```
#include <fstream>
#include <vector>
#define NMAX 1001
using namespace std;
ifstream fin("graf.in");
ofstream fout("graf.out");
int n;
vector< vector<int> > v(NMAX);
vector<bool> viz(NMAX, false);
void dfs(int x)
{viz[x]=true; fout<<x<<' ';
 for (int i=0; i<v[x].size(); i++)
    if (!viz[v[x][i]]) dfs(v[x][i]);
}
```

```

int main()
{
    int m, i, x, y, nrc=0;
    fin>>n>>m;
    for (i=0; i<m; i++)
        {fin>>x>>y;
         v[x].push_back(y);
         v[y].push_back(x); }
    for (i=1; i<=n; i++)
        if (!viz[i])
            {fout<<"Componenta conexa "<<++nrc<<'\n';
             dfs(i);
             fout<<'\n';}
    fout.close();
    return 0;
}

```

Observație

O altă implementare posibilă ar fi să alocăm static vectorul de linii, iar liniile ce conțin liste de adiacență să fie alocate dinamic (utilizând clasa `vector`). Declarația grafului reprezentat prin liste de adiacență în acest caz va fi :

```
#include <vector>
```

Această declarație alocă un vector cu `NMAX` componente de tip `vector<int>` (unde `NMAX` este numărul maxim de vârfuri din graf).

5.2. Clasa `deque`

Pentru a utiliza această clasă, trebuie să includem fișierul antet `deque`:

```
#include <deque>
```

Cuvântul *deque* (pronunțat „dec”) este un acronim pentru *double ended queue* (coadă cu două capete sau coadă cu dublă prioritate). Clasa `deque` implementează o structură de date alocată dinamic, în care se realizează eficient operații de eliminare/extragere la ambele capete. Spre deosebire de clasa `vector`, nu se garantează că elementele sunt stocate într-o zonă continuă de memorie, ci sunt stocate în „blocuri”. Acest mod de stocare permite implementarea eficientă a inserărilor și extragerilor de la ambele capete și oferă un plus de eficiență când se lucrează cu secvențe mari, în care realocările de memorie sunt costisitoare.

Iteratorii clasei `deque` sunt iteratori cu acces aleatoriu, ca și în cazul vectorilor, dar implementarea lor este mai complexă, fiind necesară parcurgerea mai multor blocuri de elemente.

Interfața clasei `deque` este extrem de asemănătoare cu cea a clasei `vector`, prin urmare vom prezenta succint funcțiile membre ale clasei.

Constructorii și destructorul

Constructorii clasei `deque` sunt similari cu constructorii clasei `vector`, fiind supraîncărcați constructorul implicit, constructorul de umplere, constructorul bazat pe un domeniu și constructorul de copiere. Destructorul distrugă obiectul, eliberând memoria alocată dinamic.

Exemple

```
| deque<int> a;
```

Creează un `deque` cu 0 elemente de tip `int`.

```
| deque<int> a(100, 0);
```

Creează un `deque` cu 100 de elemente de tip `int` și initializează cu 0 toate elementele.

```
| string s;  
| cin>>s;  
| deque<char> d(s.begin(), s.end());
```

Creează un `deque` de dimensiune egală cu dimensiunea sirului `s` și copiază în `deque` fiecare caracter din sirul `s`.

Funcțiile membre care returnează iteratori

Clasa `deque` conține funcțiile `begin()`, `end()`, `rbegin()`, `rend()` cu semnificația uzuală. Începând cu C++ 11 există și funcțiile corespondente care returnează `const_iterator` (`cbegin()`, `cend()`, `crbegin()`, `crend()`).

Funcțiile membre referitoare la dimensiune

Alături de funcțiile uzuale `size()`, `empty()` și `max_size()` în clasa `deque` sunt definite funcțiile membre publice `resize()` și, începând cu C++ 11, `shrink_to_fit()`, în mod similar cu cele din clasa `vector`. Funcțiile `capacity()` și `reserve()` nu sunt disponibile.

Accesori

Accesul la elementele unui `deque` se realizează în mod similar cu accesul la elementele unui vector cu ajutorul operatorului de indexare `[]` și a funcțiilor membre `at()`, `front()`, `back()`.

Funcțiile membre care modifică containerul deque

Ca pentru orice container, este supraîncărcat operatorul de atribuire =, funcția membră swap(), funcția clear(), funcțiile insert() și erase(). Asemănător cu cele din clasa vector sunt definite funcțiile membre push_back(), pop_back(), assign(), emplace(), emplace_back(). În plus, sunt definite următoarele funcții membre:

- push_front()

| **void push_front (const value_type& val);**

Inserează un element cu valoarea val la început (înainte de primul element).

- pop_front()

| **void pop_front();**

Elimină și distrugе primul element.

Operațiile push_front() și pop_front() se execută în timp constant.

Observație

Funcțiile insert() și erase() permit inserarea, respectiv ștergerea elementelor de oriunde din container, inclusiv din interior (în acest caz iteratorii, pointerii și referințele sunt invalidate).

Operatorii relaționali

Pentru clasa deque sunt supraîncărcați toți operatorii relaționali, compararea realizându-se din punct de vedere lexicografic.

Aplicația 6. Secvreg

Să considerăm secvențe constituite numai din paranteze rotunde și paranteze pătrate, adică din caracterele (). []. Prin definiție, o secvență de paranteze este regulată dacă se obține aplicând următoarele reguli :

1. () și [] sunt secvențe regulate ;
2. Dacă A este regulată, atunci (A) și [A] sunt secvențe regulate ;
3. Dacă A și B sunt regulate, atunci AB este secvență regulată.

De exemplu, () () [] și [(())] [] sunt regulate, în timp ce] [sau [() sau ([) nu sunt regulate.

Se consideră o secvență de paranteze. La fiecare pas se inserează o paranteză (rotundă sau pătrată, deschisă sau închisă) la începutul sau la sfârșitul secvenței.

Cerință

Scriți un program care să determine, după fiecare pas, lungimea celei mai scurte subsecvențe regulate (formată din caractere consecutive ale secvenței) care conține paranteza inserată la pasul respectiv.

Date de intrare

Fișierul de intrare `secvreg.in` conține pe prima linie secvența inițială de paranteze. Pe cea de-a doua linie a fișierului de intrare se află un număr natural N ce reprezintă numărul de pași. Fiecare dintre următoarele N linii conține un număr natural A și un caracter C separate printr-un singur spațiu. Dacă A este 0 (zero) atunci caracterul C este inserat la începutul secvenței; dacă A este 1 atunci caracterul C este inserat la sfârșitul secvenței.

Date de ieșire

Fișierul de ieșire `secvreg.out` conține N linii. Pe cea de a i-a linie va fi afișată lungimea celei mai scurte subsecvențe regulate care conține paranteza inserată la pasul i. Dacă nu există o astfel de subsecvență, pe linia i veți afișa 0.

Restricții

- Lungimea secvenței inițiale de paranteze este $\leq 100\ 000$
- $1 \leq N \leq 100\ 000$

Exemple

<code>secvreg.in</code>	<code>secvreg.out</code>	<code>secvreg.in</code>	<code>secvreg.out</code>
(]	0	[)	0
3	0	3	2
1)	2	0)	6
0)		0 (
0 (0 (

(.campion 2004)

Soluție

Vom utiliza două cozi cu dublă prioritate :

1. coada cu dublă prioritate a cu elemente de tip `char` în care inserăm succesiv caracterele din sirul inițial, apoi pe cele specificate în operațiile din fișier;
2. coada cu dublă prioritate b cu elemente de tip `int`; $b[i] =$ lungimea secvenței regulate care începe la poziția i.

Inițial inserăm în ordine toate caracterele sirului citit în coada cu dublă prioritate a. Pentru a păstra ordinea, fiecare nou caracter va fi inserat la sfârșitul cozii. Când inserăm o paranteză închisă la sfârșitul cozii, iar pe ultima poziție în a este paranteza deschisă corespunzătoare, deducem că se formează o secvență regulată cu lungimea cu 2 mai mare decât cea deja existentă pe această poziție. Prin urmare, eliminăm din cozile a, respectiv b elementele finale (adică eliminăm din a secvența regulată

formată deja), apoi actualizăm în b lungimea secvenței care începe pe ultima poziție, cumulând lungimea secvenței regulate eliminate. În caz contrar, inserăm în a la final caracterul citit, iar în b inserăm la final valoarea 0 (fiindcă aceasta este lungimea secvenței regulate formate cu caracterul citit).

Când trebuie să inserăm la începutul cozii a o paranteză deschisă, iar la începutul cozii a era paranteza închisă corespunzătoare, se formează o secvență regulată cu 2 mai mare și executăm aceleași operații ca în cazul precedent, dar la începutul cozilor cu dublă prioritate.

```
#include <fstream>
#include <deque>
using namespace std;

ifstream fin("secvreg.in");
ofstream fout("secvreg.out");
deque<char> a;
deque<int> b;
string s;

inline bool deschis(char c)
{ return c=='(' || c=='['; }
inline char pereche(char c)
{
    if (c==')') return '(';
    if (c=='(') return ')';
    if (c==']') return '[';
    if (c=='[') return ']';
}

int la_sfarsit(char c)
{ int ret;
  if ( !a.empty() && !deschis(c) && a.back()==pereche(c) )
    {ret=b.back()+2; //retin lungimea secventei regulate
     b.pop_back(); a.pop_back();
      //elimin secventa regulata formata
     b[b.size()-1]+=ret;
      //actualizez in b lungimea secventei regulate
     return ret;
    }
  a.push_back(c); b.push_back(0);
  return 0;
}

int la_inceput(char c)
{ int ret;
  if ( !a.empty() && deschis(c) && a.front()==pereche(c) )
    {ret=b.front()+2; //retin lungimea secventei regulate
     a.pop_front(); b.pop_front();
      b[0] += ret;
    }
}
```

```

        return ret;
    }
    a.push_front(c); b.push_front(0);
    return 0;
}

int main( )
{
    char c;
    int i, poz, n;
    fin>>s;
    b.push_back(0);
    for (i=0; i<s.size(); ++i) la_sfarsit(s[i]);
    fin>>n;
    for (i=0; i<n; ++i)
        {fin>>poz>>c;
        if (poz) fout<<la_sfarsit(c)<<'\n';
         else   fout<<la_inceput(c)<<'\n';
        }
    fout.close();
    return 0;
}

```

5.3. Clasa list

Pentru a utiliza această clasă trebuie să includem fișierul antet `list`:

```
#include <list>
```

Clasa `list` implementează o structură de date care permite inserarea și extragerea elementelor din structură în mod eficient, indiferent de poziția acestora, precum și parcurgerea elementelor în ambele sensuri. Ca implementare, elementele containerului `list` sunt organizate ca o listă dublu înălțuită. Iteratorii pentru containerul `list` sunt bidirecționali, ceea ce înseamnă că nu avem acces direct la elementele unei liste, dar putem parurge elementele în ambele sensuri. Un posibil dezavantaj al acestui container este memoria suplimentară necesară pentru memorarea legăturilor în listă.

Constructorii și destructorul

Constructorii clasei `list` sunt similari cu constructorii claselor `vector` și `deque` fiind supraîncărcați constructorul implicit, constructorul de umplere, constructorul bazat pe un domeniu și constructorul de copiere. Destructorul distrugе obiectul, eliberând memoria alocată dinamic. Exceptând constructorul implicit, toți constructorii și destructorul au complexitate liniară.

Exemple

```
| list<int> a;
```

Creează o listă cu 0 elemente de tip int.

```
| list<int> a(100, 0);
```

Creează o listă cu 100 de elemente de tip int și initializează cu 0 toate elementele.

```
| string s;
cin>>s;
list<char> d(s.begin(), s.end());
```

Creează o listă de dimensiune egală cu dimensiunea sirului s și copiază în listă fiecare caracter din sirul s.

Funcțiile membre care returnează iteratori

Clasa list conține funcțiile begin(), end(), rbegin(), rend() cu semnificația ușuală. Începând cu C++ 11, există și funcțiile corespondente care returnează const_iterator (cbegin(), cend(), crbegin(), crend()).

Funcțiile membre referitoare la dimensiunea listei

În clasa list sunt disponibile numai funcțiile uzuale size(), empty() și max_size().

Accesori

Este posibil accesul doar la primul și la ultimul element al listei (funcția front() returnează o referință la primul element, iar funcția back() returnează o referință la ultimul element al listei).

Funcțiile membre care modifică lista

Ca pentru orice container, sunt supraîncărcați operatorul de atribuire =, funcția membră swap(), funcția clear(), funcțiile insert() și erase(). Asemănător cu cele din clasa deque sunt definite funcțiile membre push_back(), push_front(), pop_back(), pop_front(), assign(), emplace(), emplace_back() și emplace_front().

Operațiile specifice

Clasa container list conține funcții membre publice care permit efectuarea unor operații specifice cu liste:

- `splice()`

```
void splice(iterator poz, list& x);
void splice(iterator poz, list& x, iterator i);
void splice(iterator poz, list& x, iterator prim, iterator
ultim);
```

Funcția `splice()` mută în containerul curent, la poziția indicată de iteratorul `poz`, elemente din lista `x`. Prima formă mută toate elementele din `x`, a doua formă mută doar elementul indicat de iteratorul `i`, iar a treia formă mută elementele din `x` aflate în domeniul `[prim,ultim]`. În urma acestei operații dimensiunea listei curente și a listei `x` se modifică. Funcția `splice()` se execută în timp constant pentru primele două forme, respectiv în timp liniar, funcție de dimensiunea domeniului pentru a treia formă.

- `remove()`

```
void remove (const value_type& val);
```

Funcția `remove()` elimină și distrugă în timp liniar toate elementele din lista curentă egale cu valoarea `val`.

- `remove_if()`

```
template <class Predicate>
void remove_if (Predicate pred);
```

Funcția `remove_if()` elimină și distrugă în timp liniar toate elementele din lista curentă pentru care predicatul `pred` are valoarea `true`.

- `unique()`

```
void unique();
template <class BinaryPredicate>
void unique (BinaryPredicate binary_pred);
```

După apelarea primei forme a funcției `unique()`, lista curentă nu va mai conține elemente consecutive de valori egale (duplicatele fiind eliminate). După apelarea celei de a doua forme a funcției `unique()`, lista curentă nu va mai conține elemente consecutive pentru care predicatul binar `binary_pred` să returneze valoarea `true`. Dintr-o succesiune de valori pentru care `binary_pred` returnează `true` pentru oricare două valori consecutive, se păstrează doar prima valoare, celelalte fiind eliminate. Elementele eliminate sunt distruse. Complexitatea funcției `unique()` este liniară.

- `merge()`

```
void merge (list& x);
template <class Compare>
void merge (list& x, Compare comp);
```

Realizează interclasarea listei curente cu lista x , mutând elementele din x în lista curentă pe pozițiile corespunzătoare. Evident, se consideră că inițial elementele din lista curentă și elementele din lista x sunt deja ordonate crescător considerând operatorul $<$ (pentru prima formă), respectiv ordonate conform criteriului specificat de predicatul binar comp (pentru a doua formă). Interclasarea se realizează în timp liniar.

- `sort()`

```
void sort();
template <class Compare>
void sort (Compare comp);
```

Ordonează elementele listei curente crescător considerând operatorul $<$ (pentru prima formă), respectiv considerând criteriul specificat de predicatul binar comp (pentru a doua formă), mutând elementele listei. Complexitatea algoritmului de sortare este $O(N \log N)$, unde cu N am notat numărul de elemente din listă.

- `reverse()`

```
void reverse();
```

Inversează în timp liniar ordinea elementelor din lista curentă.

Operatorii relaționali

Operatorii relaționali au fost supraîncărcați pentru clasa `forward_list`, compararea realizându-se din punct de vedere lexicografic.

Aplicația 6. Coment

Elevii de clasa a XII-a au un sistem propriu de a-și transmite comentariile la română. În urma îndelungatei lor colaborări, și-au format un sistem de relații astfel încât, oricare ar fi doi elevi, unul primește comentarii de la celălalt. Evident, orice comentariu primit poate fi transmis mai departe.

Cerință

Presupunând că în clasa a XII-a sunt N elevi, numerotați distinct de la 1 la N , și că sistemul de relații dintre elevi este cunoscut, scrieți un program care să găsească o modalitate prin care un comentariu, transmis de unul dintre elevi, să ajungă pe la toți elevii o singură dată.

Date de intrare

Fișierul de intrare `coment.in` conține pe prima linie numărul natural N , reprezentând numărul de elevi. Pe următoarele $N*(N-1)/2$ linii se află perechi x y de numere naturale distincte cuprinse între 1 și N , cu semnificația că elevul x transmite comentarii elevului y .

Date de ieșire

Fișierul de ieșire coment.out conține pe prima linie o succesiune de N numere naturale distincte cuprinse între 1 și N, reprezentând ordinea în care elevii primesc comentariul.

Restriții

- $2 \leq N \leq 100$
- Dacă există mai multe soluții, se va determina una singură.

Exemplu

coment.in	coment.out
4	2 4 3 1
1 2	
1 4	
3 1	
2 4	
3 2	
4 3	

Olimpiada Municipală de Informatică Iași, 2002

Soluție

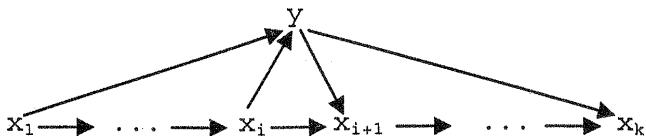
Putem asocia problemei un graf orientat în care vârfurile sunt elevii; există arc de la vârful x la vârful y dacă elevul x transmite comentarii elevului y. Din enunțul problemei deducem că graful este complet și antisimetric (oricare ar fi doi elevi x și y există exact un arc între x și y), adică este graf turneu. Problema solicită construirea unui drum hamiltonian în acest graf (drum care trece prin fiecare vârf al grafului exact o dată). Putem demonstra că în orice graf turneu există un drum hamiltonian. Demonstrația este constructivă și va sta la baza algoritmului nostru.

Vom alege un vârf oarecare din graf (de exemplul vârful 1) și construim un drum elementar maximal pornind din acest vârf. Inițial drumul este format dintr-un singur vârf $x_1=1$. Cât timp este posibil, extindem drumul în dreapta (mai exact, determinăm un vârf y care nu aparține drumului și pentru care există arc de la ultimul vârf de pe drum la y și adăugăm pe y la sfârșitul drumului). În mod similar, extindem cât timp este posibil drumul în stânga (determinăm un vârf y care nu aparține drumului și pentru care există arc de la y la primul vârf al drumului și îl inserăm pe y pe prima poziție pe acest drum).

Să notăm cu x_1, x_2, \dots, x_k drumul elementar maximal astfel creat. Există două cazuri posibile :

1. drumul creat conține toate vârfurile grafului; în acest caz ne-am atins scopul: am construit un drum hamiltonian;
2. drumul creat nu conține toate vârfurile grafului; în acest caz există un vârf y care nu aparține drumului. Referitor la acest vârf y putem face următoarele afirmații:
 - există arc de la x_1 la y (în caz contrar, drumul nu ar fi maximal, ci ar putea fi extins în stânga inserând pe y înaintea lui x_1);

- există arc de la y la x_k (în caz contrar, drumul nu ar fi maximal, deoarece am putea extinde drumul în dreapta, adăugând pe y după x_k);
- există cel puțin o pereche de vârfuri consecutive pe drum $x_i \ x_{i+1}$ astfel încât există arcul (x_i, y) și arcul (y, x_{i+1}) (în caz contrar nu am putea pleca din x_i cu arce către y și să ajungem în x_k cu arce plecând din y):



Inserăm pe y pe acest drum între x_i și x_{i+1} , obținând un drum elementar cu lungimea cu 1 mai mare; procedeul se repetă cât timp drumul nu este hamiltonian.

Vom reprezenta graful prin matrice de adiacență, deoarece numărul maxim de vârfuri din graf este relativ mic (100) și este necesar să identificăm arce atât în funcție de extremitatea lor inițială, cât și în funcție de extremitatea lor finală. Drumul va fi un obiect de tip `list` denumit `d`, deoarece sunt necesare operații de inserare eficiente în diverse poziții. Pentru a verifica eficient dacă un vârf aparține sau nu drumului curent vom utiliza un `vector<bool>` denumit `uz` (valoarea `false` semnificând faptul că vârful nu aparține drumului).

Programul implementează procedeul descris în demonstrația existenței drumului hamiltonian.

```

#include <iostream>
#include <list>
#include <vector>
#define NMaxVf 101
using namespace std;

int n;
bool G[NMaxVf][NMaxVf];
vector<bool> uz;
list<int> d;

void Citire();
void Determina();
void Scrie();

int main()
{
    Citire();
    Determina();
    Scrie();
    return 0;
}
  
```

```

void Citire()
{ifstream fin("coment.in");
int x, y, m, i;
fin>>n;
uz=vector<bool>(n+1, false);
m=n*(n-1)/2;
for (i=0; i<m; ++i)
{ fin>>x>>y; G[x][y]=true; }
fin.close();
}

void Scrie()
{ofstream fout("coment.out");
list<int>::iterator it;
for (it=d.begin(); it!=d.end(); ++it) fout<<*it<<' ';
fout<<'\n';
fout.close(); }

int cautpred(int vf)
{for (int i=1; i<=n; i++)
if (!uz[i] && G[i][vf]) return i;
return 0;
}
int cautsucc(int vf)
{for (int i=1; i<=n; i++)
if (!uz[i] && G[vf][i]) return i;
return 0;
}

void Determina()
{int x;
list<int>::iterator it;
//initializam drumul cu varful 1
uz[1]=true; d.push_back(1);
//extindem drumul catre dreapta, cat timp este posibil
do
{x=cautsucc(d.back());
if (x) {d.push_back(x); uz[x]=true;}
}
while (x);
//extindem drumul catre dreapta, cat timp este posibil
do
{x=cautpred(d.front());
if (x) {d.push_front(x); uz[x]=true;}
}
while (x);
for (x=2; x<=n; x++)
if (!uz[x]) //x nu apartine drumului
{//caut pozitie de inserare
for (it=d.begin(); G[*it][x]; ++it);
}
}

```

```

    //inserez pe x in pozitia curenta
    d.insert(it,x);
}
}

```

5.4. Clasa `forward_list`

Clasa `forward_list` a fost introdusă în *STL* începând cu *C++ 11* și, pentru a o utilize, trebuie să includem fișierul antet `forward_list`:

```
#include <forward_list>
```

Clasa `forward_list` implementează o structură de date care permite inserarea și extragerea elementelor în timp constant, indiferent de poziția acestora. Ca implementare, elementele containerului `forward_list` sunt organizate ca o listă simplu înlănuțită. Cu alte cuvinte, spre deosebire de `list`, unde fiecare element conținea două legături (una către următorul element din listă și una către precedentul element din listă), în `forward_list` se reține doar o singură legătură, către următorul element din listă. Prin urmare, se reduce dimensiunea memoriei suplimentare necesară pentru memorarea legăturilor.

Iteratorii containerului `forward_list` sunt iteratori de înaintare (*forward iterators*), prin urmare parcurgerea elementelor este posibilă doar într-un singur sens (de la început către sfârșit).

Constructorii, destructorul și operatorul de atribuire = sunt similari clasei `list`. Vom evidenția în continuare diferențele specifice clasei `forward_list`, pentru diferitele categorii de funcții.

Funcțiile membre referitoare la dimensiunea listei

Containerul `forward_list` este singurul container standard care, din motive de eficiență, nu dispune de funcția `size()`. Astfel, nu mai este necesară reținerea și actualizarea permanentă a dimensiunii curente a listei. Singurele funcții disponibile referitoare la dimensiunea containerului sunt `empty()` și `maxsize()`. Pentru a afla numărul de elemente din listă se poate utiliza algoritmul `distance`. De exemplu, dacă `FL` este un obiect de tip `forward_list`, pentru a determina în variabila `nr` numărul de elemente din lista `FL` putem apela funcția `distance` având ca parametri începutul și sfârșitul listei:

```
nr=distance(FL.begin(), FL.end());
```

Accesori

Este posibil accesul doar la primul element al listei utilizând funcția `front()`, care returnează o referință la primul element.

Funcțiile membre care returnează iteratori

Clasa `forward_list` conține funcțiile uzuale `begin()`, `end()`, `cbegin()` și `cend()` cu semnificația uzuală. În plus, sunt definite două funcții denumite `before_begin()`, respectiv `cbefore_begin()` care returnează un iterator, respectiv `const_iterator` care indică poziția dinaintea primului element din listă. Evident, acest iterator nu poate fi dereferențiat, ci poate fi folosit pentru a indica această poziție în apeluri ale funcțiilor `emplace_after()`, `insert_after()`, `erase_after()` sau `splice_after()`.

Funcțiile membre care modifică lista

Asemănător clasei `list` sunt definite funcțiile membre `push_front()`, `pop_front()`, `assign()`, `emplace()`, `emplace_front()`, `resize()`, `swap()` și `clear()`. Următoarele funcții sunt specifice clasei `forward_list`:

- `insert_after()`

```
iterator insert_after(const_iterator poz,
                      const value_type& val);
iterator insert_after(const_iterator poz, size_type n,
                      const value_type& val);
template <class InputIterator>
iterator insert_after(const_iterator poz,
                      InputIterator prim, InputIterator ultim);
iterator insert_after(const_iterator poz,
                      initializer_list<value_type> il);
```

Funcția `insert_after()` inserează în lista curentă, după elementul indicat de iteratorul `poz`, un element cu valoarea `val` (prima formă), `n` elemente cu valoarea `val` (a doua formă), copii ale elementelor din domeniul `[prim,ultim)` (a treia formă) sau copii ale elementelor din lista de inițializare `li` (ultima formă). Funcția returnează ca rezultat un iterator care indică poziția primului element inserat.

- `erase_after()`

```
iterator erase_after (const_iterator poz);
iterator erase_after (const_iterator poz,
                      const_iterator ultim);
```

Funcția `erase_after()` elimină din listă elementul de după cel indicat de iteratorul `poz` (prima formă), respectiv elementele din domeniul `(poz,ultim)`.

Funcția returnează un iterator care indică poziția elementului care urmează după ultimul element eliminat.

- `emplace_after()`

```
template <class... Args>
iterator emplace_after(const_iterator poz, Args&&... p);
```

Funcția `emplace_after()` construiește un element (utilizând constructorul corespunzător parametrilor specificați după iteratorul `poz`) și inserează elementul creat după cel indicat de `poz`.

Observație

Funcțiile `insert_after()` și `emplace_after()` pot fi utilizate și pentru inserarea unui element la începutul listei, dacă iteratorul `poz` indică poziția precedență primului element din listă (iteratorul returnat de funcția `before_begin()`). În mod similar poate fi șters din listă primul element cu funcția `erase_after()`.

Operațiile specifice

Clasa container `forward_list` conține funcțiile membre publice `remove()`, `remove_if()`, `sort()`, `reverse()`, `unique()`, `merge()` cu efect similar celor din clasa `list`. Funcția `splice()` a fost înlocuită cu funcția `splice_after()`:

- `splice_after()`

```
void splice_after(const_iterator poz, forward_list& FL);
```

```
void splice_after (const_iterator poz, forward_list& FL,
                   const_iterator i);
```

```
void splice_after (const_iterator poz, forward_list& FL,
                   const_iterator prim, const_iterator ultim);
```

Funcția `splice_after()` mută în containerul curent, după elementul indicat de iteratorul `poz`, elemente din lista `FL`. Prima formă mută toate elementele din `FL`, a doua formă mută doar elementul indicat de iteratorul `i`, iar a treia formă mută elementele din `FL` aflate în domeniul `[prim,ultim]`. În urma acestei operații dimensiunea listei curente și a listei `FL` se modifică.

Observație

Deoarece nu este necesară parcurgerea listei în ambele sensuri, pentru implementarea soluției problemei *Coment* am fi putut utiliza și o listă simplu înlățuită (`forward_list`). Desigur, ar fi trebuit să reținem permanent poziția precedentă celei în care realizăm inserarea, ceea ce ar fi condus la un cod mai puțin elegant.

5.5. Clasa `array`

Clasa `array` a fost introdusă începând cu `C++ 11` și poate fi utilizată dacă includem fișierul antet `array`:

```
#include <array>
```

Această clasă implementează o succesiune de dimensiune fixă formată din elemente de același tip, stocate într-o zonă continuă de memorie. Această structură este foarte asemănătoare cu tablourile unidimensionale din limbajul C, dar dispune de o serie de funcții membre care fac posibilă utilizarea acestora în mod similar cu celealte containere standard.

Spre deosebire de clasa `vector`, memoria alocată elementelor unui `array` este fixă, nu se alocă dinamic. Prin urmare, dimensiunea unui `array` nu poate fi modificată dinamic și nu este necesar să fie stocată intern.

Declararea unui obiect de tip `array`:

```
array<Tip, Dim> ob;
```

unde `Tip` reprezintă tipul comun al elementelor, iar `Dim` dimensiunea fixă (numărul de elemente).

De exemplu, pentru a declara un `array` a cu 10 elemente de tip `int`:

```
array<int, 10> a;
```

Funcțiile membre care returnează iteratori

Clasa `array` conține funcțiile `begin()`, `end()`, `rbegin()`, `rend()`, `cbegin()`, `cend()`, `crbegin()`, `crend()`, cu semnificația uzuală.

Funcțiile membre referitoare la dimensiune

Sigurele funcții membre referitoare la dimensiune sunt `size()`, `empty()` și `max_size()`. Funcțiile `size()` și `max_size()` returnează aceeași valoare, dimensiunea fixă specificată la declarare.

Accesori

Accesul la elementele unui `array` se realizează în mod similar cu accesul la elementele unui `vector`, cu ajutorul operatorului de indexare `[]` și a funcțiilor membre `at()`, `front()`, `back()`. Este de asemenea implementată funcția membră `data()` care returnează un pointer către începutul zonei de memorie care conține elementele containerului.

Funcțiile membre care modifică containerul array

Ca pentru orice container, este supraîncărcată funcția membră `swap()` care interschimbă în timp liniar elementele a două containere. În plus, este definită funcția membră `fill()`:

```
| void fill (const value_type& val);
```

Atribuie tuturor elementelor din container valoarea `val`.

Operatorii relationali

Pentru clasa `array` sunt supraîncărcăți toți operatorii relationali, compararea realizându-se din punct de vedere lexicografic.

5.6. Exerciții și probleme propuse

1. Ce va afișa pe ecran următoarea secvență de instrucțiuni:

```
| vector<char> v1;
vector<char> v2(100);
v1=v2;
v1[0]='s';
cout<<v1.size()<<' '<<v1.capacity()<<'\n';
v1.resize(10);
cout<<v1.size()<<' '<<v1.capacity()<<'\n';
```

- | | |
|--|--|
| a. Nimic, deoarece această secvență produce eroare la execuție (<code>v1</code> este vid, prin urmare nu ne putem referi la <code>v1[0]</code>); | b. |
| c.
100 100
10 100; | 0 0
10 10;
d.
100 100
10 10. |

2. Care dintre următoarele secvențe creează un vector cu 100 componente reale:

- | | |
|---|--|
| a. <code>vector<float> v1(100);</code> | b. <code>vector<float> v1;</code>
<code>v1.reserve(100);</code> |
| c. <code>vector<float> v1;</code>
<code>v1.resize(100);</code> | d. <code>vector<float> v1;</code>
<code>v1=vector<float>(100);</code> |

3. Ce se va afișa pe ecran după executarea următoarei secvențe de instrucțiuni, dacă de la tastatură se introduce Mama are mere.

```
| string s;
cin>>s;
vector<char> v1(s.begin()+1, s.end());
cout<<v1.size()<<' '<<v1.capacity()<<'\n';
```

```

v1.insert(v1.begin() + 2, s.begin(), s.end());
cout << v1.size() << ' ' << v1.capacity() << '\n';
v1.erase(v1.end() - 1);
cout << v1.size() << ' ' << v1.capacity() << '\n';
for (int i = 0; i < v1.size(); i++) cout << v1[i];

```

4. Presupunând că este necesară o structură de date secvențială care să permită inserări și extrageri de elemente doar de la începutul sau sfârșitul secvenței, care container reprezintă cea mai eficientă implementare? Justificați răspunsul.

- a. vector;
b. deque;
c. list;
d. Niciunul dintre acestea

5. Care dintre următoarele secvențe afișează elementele listei v?

```
| list<int> v;
```

2.

```
for (int i=0; i<v.size(); ++i)
    cout<<v[i]<<' ';
```

b.

```
list<int>::iterator it;
for (it=0; it<v.size(); ++it)
    cout<<*it<<';'
```

C₆

```
list<int>::iterator it;
for (it=v.begin(); it<v.end(); ++it)
    cout<<*it<<endl;
```

d

```
list<int>::iterator it;
for (it=v.begin(); it!=v.end(); ++it)
    cout<*it<<! ' ';
```

6. Ce elemente contin listele v1 și v2 după executarea următoarei secvențe:

```
int i;
list<int> v1;
for (i=1; i<5; ++i) v1.push_back(i);
list<int> v2;
for (i=1; i<9; i+=2) v2.push_back(i);
v1.merge(v2);
```

7. Ce elemente contin listele v1 și v2 după executarea următoarei secvențe:

```
int i;
list<int> v1;
for (i=1; i<5; ++i) v1.push_back(i);
list<int> v2;
for (i=1; i<8; ++i) v2.push_back(i);
```

```
v2.reverse();
list<int>::iterator it1=v1.begin(), it2=v2.begin();
it1++; it2++;
v1.splice(it1, v2, it2, v2.end());
```

8. Ce se va afișa pe ecran în urma execuției următoarei secvențe:

```
deque<int> d;
for (int i=1; i<5; ++i) d.push_back(i);
deque<int>::iterator it=d.begin()+2;
d.erase(it);
cout<<*it<< ' ';
```

9. Considerând următoarea definiție a funcției lafel():

```
bool lafel(char c1, char c2)
{string s("aeiouAEIOU");
 return s.find(c1)<string::npos&&s.find(c2)<string::npos; }
```

Ce conține lista d după executarea următoarei secvențe?

```
string s("Mamaia si Aida vorbeau neaos");
list<char> d(s.begin(), s.end());
d.unique(lafel);
```

10. Pentru care dintre următoarele declarații, construcția `it+1` este validă?

- a. `vector<char>::iterator it;`
- b. `list<double>::iterator it;`
- c. `deque<int>::iterator it;`
- d. Niciuna dintre variantele precedente.

11. Care dintre următoarele instrucțiuni elimină primul element din containerul `deque<int> d`:

- | | |
|-------------------------------------|-------------------------------------|
| a. <code>d.pop_front();</code> | b. <code>d.erase(d.front());</code> |
| c. <code>d.erase(d.begin());</code> | d. <code>d.pop();</code> |

12. Considerând lista `list<char> d`, utilizați funcția membră `unique()` pentru a elimina din listă spațiile redundante.

13. Considerând vectorul `vector<int> v`, scrieți o secvență de instrucțiuni care să eliminate toate elementele pare din vector. Realizați aceeași operație pentru lista `list<int> d`, într-o manieră diferită. Pentru care dintre cele două containere eliminarea se realizează mai eficient?

14. Utilizând clasa specializată `vector<bool>`, implementați operațiile cu mulțimi de numere naturale (`VMAX` (reuniune, intersecție, diferență, diferență simetrică). Comparați operațiile astfel implementate cu varianta ce utilizează `vector<int>` și algoritmii standard de lucru cu mulțimi din *STL*.

15. *Arme*

Vasile joacă (din nou!) jocul său preferat cu împușcături. Personajul său are la brâu N arme, așezate în N huse speciale, numerotate de la 1 la N . Arma din husa i are puterea pb_i ($1 \leq i \leq N$).

În camera armelor a găsit M arme, așezate pe perete, în M locații, numerotate de la 1 la M. Pentru fiecare armă j ($1 \leq j \leq M$) este cunoscută puterea sa pc_j .

Vasile poate înlocui arme pe care le are la brâu cu arme aflate pe perete în camera armelor. La o înlocuire el ia arma de pe perete din locația j ($1 \leq j \leq M$) și o pune la brâu în husa i ($1 \leq i \leq N$), iar arma din husa i o pune pe perete în locația j.

Cerință

Scrieți un program care să determine suma maximă a puterilor armelor pe care le va avea la brâu Vasile după efectuarea înlocuirilor.

Date de intrare

Fișierul de intrare `arme.in` conține pe prima linie numerele naturale N M, reprezentând numărul de arme pe care le are la brâu, respectiv numărul de arme aflate în camera armelor. Pe a doua linie se află N numere naturale $pb_1 pb_2 \dots pb_N$ reprezentând în ordine puterile armelor pe care Vasile le are la brâu. Pe a treia linie se află M numere naturale $pc_1 pc_2 \dots pc_M$ reprezentând în ordine puterile armelor aflate în camera armelor. Numerele scrise pe aceeași linie sunt separate prin spațiu.

Date de ieșire

Fișierul de ieșire `arme.out` va conține o singură linie pe care va fi scrisă suma maximă a puterilor armelor de la brâul lui Vasile, după efectuarea înlocuirilor.

Restricții

- $1 \leq N, M \leq 1000$
- Puterile armelor sunt numere naturale ≤ 10000 .

Exemple

<code>arme.in</code>	<code>arme.out</code>
3 2 3 1 7 4 5	16

Olimpiada Județeană de Informatică, 2012

16. Segm

Să considerăm N segmente închise situate pe axa OX. Pentru fiecare segment se cunosc extremitatea inițială x și extremitatea finală y.

Cerință

Să se determine o mulțime formată dintr-un număr minim de puncte distincte de abscise întregi situate pe axa OX, cu proprietatea că orice segment conține cel puțin două puncte din mulțime.

Date de intrare

Fișierul de intrare `segm.in` conține pe prima linie numărul natural N. Pe următoarele N linii sunt descrise cele N segmente, câte un segment pe o linie. Pentru

fiecare segment sunt specificate două numere naturale separate prin spațiu x y reprezentând extremitățile segmentului.

Date de ieșire

Fișierul de ieșire `segm.out` va conține două linii. Pe prima linie este scris un număr natural `MIN`, reprezentând numărul minim de puncte din mulțimea determinată. Pe cea de a doua linie sunt scrise `MIN` numere naturale în ordine strict crescătoare, separate prin spațiu, reprezentând abscisele punctelor din mulțime.

Restricții

- $1 \leq N \leq 3000$
- $0 < x < y < 10^6$
- Dacă există mai multe soluții, afișați oricare dintre acestea.

Exemple

<code>segm.in</code>	<code>segm.out</code>
5	5
1 10	5 10 12 23 24
10 12	
1 10	
1 10	
23 24	

.campion 2009

17. Compus

Un sir de numere se numește *compus* dacă îndeplinește simultan condițiile :

- a. conține un număr impar de elemente ;
- b. elementul de la mijlocul sirului este mai mare sau egal cu toate elementele care îl precedă și mai mic sau egal cu toate elementele care îl succedă.

De exemplu, sirul $(1, 6, 2, 8, 18, 28, 9)$ este compus.

Secvența $a_1, a_2, a_3, \dots, a_i$ se numește prefix de lungime i al sirului a .

Cerință

Să se determine lungimea celui mai lung prefix compus al unui sir de numere.

Date de intrare

Fișierul de intrare `compus.in` conține pe prima linie un număr natural `N` reprezentând numărul de valori din sir. Pe cea de a doua linie sunt scrise `N` numere întregi separate prin câte un spațiu, reprezentând sirul.

Date de ieșire

Fișierul de ieșire `compus.out` va conține o singură linie, pe care va fi scris un singur număr natural reprezentând lungimea maximă a unui prefix compus.

Restricții

- $1 \leq N \leq 1\ 000\ 000$
- $0 \leq a_i \leq 10^9$, pentru orice $1 \leq i \leq N$

Exemplu

compus.in	compus.out	Explicații
10 1 4 3 8 10 8 9 7 2 11	7	Există două prefixe compuse: (1) și (1, 4, 3, 8, 10, 8, 9).

.campion 2008

18. Secvop

Fie N un număr natural și A un vector. Inițial $A[i] = i$, pentru orice $i=1, 2, \dots, N$. Asupra acestui vector se pot aplica operații de două tipuri:

Operație	Efect
I St Dr	Se inversează subsecvența din vector care începe la poziția St și se termină la poziția Dr ($1 \leq St \leq Dr \leq N$). De exemplu, pentru $N=10$, aplicând operația I 2 7 obținem vectorul 1 7 6 5 4 3 2 8 9 10;
S St Dr	Se afișează suma elementelor din subsecvența care începe pe poziția St și se termină pe poziția Dr ($1 \leq St \leq Dr \leq N$). De exemplu, pentru $N=10$, aplicând S 2 7 se afișează 27.

Cerință

Scrieți un program care să execute o secvență dată de operații.

Date de intrare

Fișierul de intrare secvop.in conține pe prima linie două numere naturale separate prin spațiu N și K (unde N este dimensiunea vectorului, iar K este numărul de operații ce trebuie să fie executate). Pe următoarele K linii sunt scrise K operații, câte o operație pe o linie. Aceste operații sunt scrise în ordinea în care trebuie să fie executate. O operație este descrisă printr-un caracter (I sau S) urmat de spațiu, apoi de indicele St , apoi de spațiu, apoi de indicele Dr ($1 \leq St \leq Dr \leq N$).

Date de ieșire

Fișierul de ieșire secvop.out va conține atâtea linii câte operații de tipul 2 (operații de scriere) se află în fișierul de intrare. Pe linia i va fi scris rezultatul afișat de cea de-a i -a operație de scriere din fișierul de intrare.

Restricții

- $1 \leq K \leq 5000$
- $0 < N \leq 10^9$
- În fișierul de intrare se află cel puțin o operație de scriere.

Exemplu

<i>secvop.in</i>	<i>secvop.out</i>	<i>Explicație</i>
15 4 S 2 11 I 10 15 I 1 10 S 5 10	65 21	Vectorul inițial: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 S 2 11 afișează suma elementelor de la poziția 2 la poziția 11: $2+3+4+5+6+7+8+9+10+11=65$ I 10 15 inversează elementele din subsecvența de la poziția 10 la poziția 15. Se obține vectorul: 1 2 3 4 5 6 7 8 9 15 14 13 12 11 10 I 1 10 inversează elementele din subsecvența de la poziția 1 la poziția 10. Se obține vectorul: 15 9 8 7 6 5 4 3 2 1 14 13 12 11 10 S 5 10 afișează suma elementelor de la poziția 5 la poziția 10 inclusiv: $6+5+4+3+2+1=21$

.campion 2008

19. Chei

Vasile își păstrează economiile în N purceluși-pușculită de ceramică neagră. Purcelușii sunt numerotați de la 1 la N. Fiecare purceluș are un capac care poate fi deschis cu cheia corespunzătoare. În purceluș Vasile a pus bani, dar și cheile de la purceluș. Desigur, Vasile ține minte în care purceluș a pus cheile.

Fiindcă vrea să-și cumpere o mașină nouă, Vasile are nevoie de toți banii din purceluș. Pentru a obține banii dintr-un purceluș el poate sparge purcelușul sau poate deschide capacul cu cheia (dacă o are).

Cerință

Cunoscând distribuția cheilor în purceluș, să se determine numărul minim de purceluș pe care trebuie să îi spargă Vasile astfel încât să poată obține toți banii.

Date de intrare

Fișierul de intrare *chei.in* conține pe prima linie un număr natural N, reprezentând numărul de purceluș. Pe fiecare dintre următoarele N linii este scris câte un număr natural cuprins între 1 și N; numărul de pe linia i+1 reprezintă numărul purcelușului în care este plasată cheia de la purcelușul i.

Date de ieșire

Fișierul de ieșire *chei.out* va conține o singură linie pe care va fi scris un singur număr natural reprezentând numărul minim de purceluș ce trebuie să fie spart pentru a obține toți banii.

Restricție

- $1 \leq N \leq 100000$

Exemple

<i>chei.in</i>	<i>chei.out</i>	<i>Explicație</i>
4 2 1 2 4	2	Cheile de la purcelușii 1 și 3 se află în purcelușul 2. Cheia de la purcelușul 2 se află în purcelușul 1. Cheia de la purcelușul 4 se află în purcelușul 4. Numărul minim de purceluș ce trebuie să fie spart este 2 (spargem purcelușul 2 și purcelușul 4).
4 1 2 3 4	4	Numărul minim de purceluș ce trebuie să fie sparți este 4.

.campion 2010

6. Clasele adaptor

Clasele adaptor pentru containere sunt definite pe baza unei alte clase container, asigurând o funcționalitate restrânsă pentru aceasta. Există trei clase adaptor predefinite în *STL*, bazate pe containere secvențiale: *queue* (coada), *stack* (stiva), și *priority_queue* (coada cu prioritate).

6.1. Clasa adaptor *queue* (coada)

Clasa adaptor *queue* este definită pentru a realiza operațiile specifice unei structuri de date de tip **FIFO** (First In First Out). Într-o astfel de structură – denumită *coadă* – elementele sunt inserate la un capăt al acesteia (sfărșitul cozii) și extrase de la celălalt capăt (începutul cozii).

Pentru a utiliza această clasă trebuie să includem fișierul antet *queue*:

```
#include <queue>
```

Declarația clasei adaptor *queue*:

```
template <class T, class Container=deque<T> > class queue;
```

Observați că această clasă şablon are doi parametri:

- *T* – reprezintă tipul elementelor containerului ;
- *Container* – reprezintă clasa container care este utilizată pentru memorarea elementelor ; implicit aceasta este clasa *deque*.

Constructorii și destructorul

Pentru clasa adaptor *queue* sunt definiți constructorul implicit, constructorul de copiere și destructorul, care distrugе obiectul, eliberând memoria alocată dinamic.

Exemple

```
queue<int> C;
```

Creează o coadă denumită C cu 0 elemente de tip int. Containerul utilizat pentru stocarea elementelor cozii este cel implicit, adică deque.

| queue<pair<int, int> > Cp;

Creează o coadă denumită Cp cu 0 elemente de tip pair<int, int>. Containerul utilizat pentru stocarea elementelor cozii este cel implicit.

| queue<string, list<string> > Csir;

Creează o coadă denumită Csir cu 0 elemente de tip string. Containerul utilizat pentru stocarea elementelor cozii este list<string>.

**| string sir = "gigel";
deque<char> d(sir.begin(), sir.end());
queue<char> Cc(dcoada);**

Am creat o coadă denumită Cc cu elemente de tip char, pe care am inițializat-o cu conținutul containerului deque d. Inițializarea a fost posibilă deoarece a fost utilizat constructorul de copiere al clasei deque, care este clasa container suport, în acest caz.

Funcțiile membre referitoare la dimensiunea containerului

Sunt definite funcțiile uzuale size() și empty(). Funcția size() returnează numărul de elemente din coadă, iar funcția empty() returnează valoarea logică true în cazul în care coada este vidă.

Accesori

Pentru o coadă este permis accesul doar la primul și la ultimul element din coadă: funcțiile membre front() și back() returnează o referință la primul, respectiv la ultimul element al cozii.

Funcțiile membre care modifică containerul

Având în vedere faptul că această clasă implementează structura de date de tip FIFO, sunt definite doar două funcții membre care modifică containerul:

- push()

| void push (const value_type& val);

Inserează un element cu valoarea val la sfârșitul cozii.

- pop()

| void pop();

Elimină și distrugе primul element din coadă.

Începând cu C++ 11 este definită și funcția membru `emplace()` care construiește și inserează un element la sfârșitul cozii, precum și funcția membră `swap()` care interschimbă coada curentă cu cea specificată ca parametru.

Operatorii relaționali

Pentru clasa `queue` sunt supraîncărcați toți operatorii relaționali, compararea realizându-se din punct de vedere lexicografic.

Observații

1. Toate operațiile asupra containerului se execută în timp constant.
2. Începând cu C++ 11 este definită și funcția independentă `swap()` care schimbă între ele conținutul celor două containere `queue` indicate ca parametri.

Aplicația 1. Magic

Jocul Tabla Magică este compus din opt piese pătrate, numerotate cu valorile de la 1 la 8, dispuse pe două linii, similar unui tablou bidimensional cu 2 linii și 4 coloane. O configurație posibilă este cea din figura următoare :

1	4	5	8
6	3	2	7

Sunt posibile modificări ale configurației prin utilizarea următoarelor mutări :

- Mutarea A: Pătratele de pe pozițiile $T[1][1]$, $T[1][2]$, $T[2][1]$, $T[2][2]$ se rotesc în sensul acelor de ceasornic.
- Mutarea B: Pătratele de pe pozițiile $T[1][2]$, $T[1][3]$, $T[2][2]$, $T[2][3]$ se rotesc în sensul acelor de ceasornic.
- Mutarea C: Pătratele de pe pozițiile $T[1][3]$, $T[1][4]$, $T[2][3]$, $T[2][4]$ se rotesc în sensul acelor de ceasornic.
- Mutarea D: Pătratele de pe pozițiile $T[1][1]$ și $T[2][1]$ se interschimbă.

Cerință

Scriți un program care să determine numărul minim de mutări necesare pentru a ajunge de la o configurație inițială la o configurație finală.

Date de intrare

Fișierul de intrare `magic.in` conține pe primele două linii câte 4 valori reprezentând configurația inițială, iar pe următoarele două linii câte 4 valori reprezentând configurația finală.

Date de ieșire

Fișierul `magic.out` va conține o singură linie pe care se va scrie numărul minim de mutări necesare pentru a transforma configurația inițială în configurația finală.

Restricție

- Întotdeauna există o succesiune de mutări care duce la configurația finală.

Exemplu

magic.in	magic.out	Explicații
1 2 3 4	2	1 2 3 4 A 5 1 3 4 D 6 1 3 4
5 6 7 8		5 6 7 8 ==> 6 2 7 8 ==> 5 2 7 8
6 1 3 4		
5 2 7 8		

(.campion 2007)

Soluție

Vom utiliza o coadă pentru a reține toate configurațiile posibile care se pot obține pornind de la configurația inițială.

La început, coada va conține doar configurația inițială. Algoritmul de generare constă în aplicarea următorilor pași :

- Extragem din coadă configurația curentă ;
- Pentru fiecare dintre cele 4 mutări posibile generăm noua configurație. Dacă această nouă configurație nu a fost generată anterior, atunci o inserăm în coadă, marcând faptul că am introdus-o în coadă și reținem totodată încă o configurație parcursă în plus, adică numărul de mutări necesare pentru obținerea acestei noi configurații.

Algoritmul se oprește atunci când extragem din coadă configurația finală, acest lucru fiind întotdeauna posibil.

Pentru a verifica dacă o configurație este deja în coada C, am definit funcția `perm_ord()`, care asociază oricărei configurații un număr natural unic. Ne bazăm pe faptul că orice configurație este de fapt o permutare a mulțimii {1, 2, 3, 4, 5, 6, 7, 8}, numărul total al configurațiilor posibile fiind $8!$ (40320). Funcția `perm_ord()` asociază unei permutări numărul de ordine al acesteia dacă am genera permutările mulțimii {1, 2, 3, 4, 5, 6, 7, 8} în ordine lexicografică.

Pentru a reține mulțimea configurațiilor deja generate utilizăm un `vector<bool>` viz (`viz[x] = true` când configurația cu numărul de ordine x a fost deja generată).

Programul complet arată astfel :

```
#include <cstdio>
#include <queue>
#include <algorithm>
#include <vector>
#define Nmax 10
#define InFile "magic.in"
#define OutFile "magic.out"
using namespace std;

struct Element {
    int conf, nr;
} st, fin;
```

```

queue <Element> Q;
vector <bool> viz(50000, false);
int ST[Nmax], FIN[Nmax], f[Nmax];

void perm_ord (int p[], int &nr)
{ int i, j, pwr[Nmax];
nr = 0;
for (i = 0; i < Nmax; i++) pwr[i] = 0;
for (i = 1; i <= 8; i++)
{ nr += (p[i] - pwr[p[i]] - 1) * f[8-i];
  for (j = p[i] + 1; j <= 8; j++) pwr[j]++;
}
}

void ord_perm (int ord, int p[])
{ int i, j, a, nr, pwr[Nmax], v[Nmax];
for (i = 0; i < Nmax; i++) pwr[i]=v[i]=0;
for (i = 1; i <= 8; i++)
{ a=ord/f[8-i];
  nr=0;
  for (j = 0; j < 8; j++)
    if (!v[j])
      {if (nr == a)
        { p[i]=j+1, v[j]=1; break; }
      nr++; }
  ord -= (p[i]-pwr[p[i]]-1)*f[8-i];
  for (j=p[i]+1; j<=8; j++) pwr[j]++;
}
}

void copy (int A[], int B[])
{ for (int i = 1; i <= 8; i++) A[i]=B[i]; }

int calcul()
{Element x, y;
int t, X[Nmax], Y[Nmax];
Q.push(st); viz[st.conf] = true;
while (!Q.empty())
{ x = Q.front(); Q.pop(); //extrag configuratia curenta
  if (x.conf == fin.conf) return x.nr; //am gasit
  else
    {ord_perm (x.conf, X);
     y.nr = x.nr + 1;
     //miscare tip A
     copy (Y, X);
     t=Y[2]; Y[2]=Y[1],Y[1]=Y[5],Y[5]=Y[6]; Y[6]=t;
     perm_ord (Y, y.conf);
     if (!viz[y.conf])
       { Q.push (y); viz[y.conf] = true; }
     //miscare tip B
    }
}
}

```

```

copy (Y, X);
t=Y[3]; Y[3]=Y[2],Y[2]=Y[6],Y[6]=Y[7]; Y[7]=t;
perm_ord (Y, y.conf);
if (!viz[y.conf])
    { Q.push (y); viz[y.conf] = true; }
//miscare tip C
copy (Y, X);
t=Y[4]; Y[4]=Y[3],Y[3]=Y[7],Y[7]=Y[8]; Y[8]=t;
perm_ord(Y, y.conf);
if (!viz[y.conf])
    { Q.push (y); viz[y.conf] = true; }
//miscare tip D
copy (Y, X);
swap (Y[1], Y[5]), perm_ord (Y, y.conf);

if (!viz[y.conf])
    { Q.push (y); viz[y.conf] = true; }
}
}
return 0;
}

int main()
{int i;
freopen (InFile, "r", stdin);
freopen (OutFile, "w", stdout);
for (f[0]=i=1; i<Nmax; ++i) f[i]=f[i-1]*i;
for (i=1; i<=8; ++i) scanf ("%d", &ST[i]);
for (i=1; i<=8; ++i) scanf ("%d", &FIN[i]);
perm_ord(ST, st.conf); perm_ord(FIN, fin.conf);
printf ("%d\n", calcul());
return 0;
}
}

```

Aplicația 2. Alee

Parcul orașului a fost neglijat mult timp, astfel că acum toate aleile sunt distruse. Prin urmare, anul acesta Primăria și-a propus să facă reamenajări. Parcul are forma unui pătrat cu latura de n metri și este înconjurat de un gard care are exact două porți. Proiectanții de la Primărie au realizat o hartă a parcului și au trasat pe hartă un caroiaj care împarte parcul în $n \times n$ zone pătrate cu latura de 1 metru. Astfel harta parcului are aspectul unei matrice pătratice cu n linii și n coloane. Liniile și respectiv coloanele sunt numerotate de la 1 la n . Elementele matricei corespund zonelor pătrate de latură 1 metru. O astfel de zonă poate să conțină un copac sau este liberă. Edilii orașului doresc să paveze cu un număr minim de dale pătrate cu latura de 1 metru zonele libere (fără copaci) ale parcului, astfel încât să se obțină o alee continuă de la o poartă la alta.

Cerință

Scrieți un program care să determine numărul minim de dale necesare pentru construirea unei alei continue de la o poartă la cealaltă.

Date de intrare

Fișierul de intrare `alee.in` conține pe prima linie două valori naturale n și m , separate printr-un spațiu, reprezentând dimensiunea parcului, respectiv numărul de copaci care se găsesc în parc. Fiecare dintre următoarele m linii conține câte două numere naturale x și y , separate printr-un spațiu, reprezentând pozițiile copacilor în parc (x reprezintă linia, iar y reprezintă coloana zonei în care se află copacul). Ultima linie a fișierului conține patru numere naturale x_1 y_1 x_2 y_2 , separate prin câte un spațiu, reprezentând pozițiile celor două porți (x_1 și y_1 reprezintă linia și respectiv coloana zonei ce conține prima poartă, iar x_2 și y_2 reprezintă linia și respectiv coloana zonei ce conține cea de-a două poartă).

Date de ieșire

Fișierul de ieșire `alee.out` va conține o singură linie pe care va fi scris un număr natural care reprezintă numărul minim de dale necesare construirii aleii.

Restricții

- $1 \leq n \leq 175$
- $1 \leq m < n^2$
- Aleea este continuă dacă oricare două plăci consecutive au o latură comună.
- Aleea începe cu zona unde se găsește prima poartă și se termină cu zona unde se găsește cea de-a două poartă.
- Pozițiile porților sunt distințe și corespund unor zone libere.
- Pentru datele de test există întotdeauna soluție.

Exemplu

<code>alee.in</code>	<code>alee.out</code>	<i>Explicații</i>
8 6	15	O modalitate de a construi aleea cu număr minim de dale este :
2 7		OOO -----
3 3		-- OO --x--
4 6		-- xO -----
5 4		--- OOx -
7 3		--- xO ---
7 5		---- OO -
1 1 8 8		-- x - xOO - ----- OO (cu X am marcat copaci, cu - zonele libere, iar cu O dalele aleii).

(Olimpiada Județeană de Informatică, Iași, 2007)

Solutie

Este o problemă „clasică” a cărei rezolvare se bazează pe algoritmul lui Lee¹.

Vom utiliza o matrice A în care inițial vom reține valoarea 0 pentru zonele libere, respectiv valoarea -1 pentru zonele în care se află un copac.

Ulterior, pe măsură ce explorăm matricea în scopul determinării unei alei de lungime minimă vom reține în matrice pentru pozițiile explorate un număr natural care reprezintă distanța minimă de la poziția primei porți la poziția respectivă (exprimată în numărul de dale necesare pentru construirea aleii).

Pozиїile din matrice care au fost explorate le vom reține într-o coadă, în ordinea explorării lor. Deoarece identificarea unei poziții se face prin două coordonate (linie, coloană), un element al cozii va fi format dintr-o pereche, deci coada este definită astfel :

```
queue<pair<int, int> > C;
```

Vom parcurge matricea începând cu poziția primei porți.

```
a[ix][iy] = 1;           //prima dala
C.push(make_pair(ix, iy)); //plasez in coada
```

La fiecare pas vom extrage din coadă o poziție

```
pair<int, int> xx;
xx = C.front();          //determin primul element din coada
C.pop();                 //elimin primul element din coada
```

Vom explora toți cei patru vecini liberi neexplorați ai poziției extrase și îi vom introduce în coadă. Când explorăm un vecin, reținem în matricea A pe poziția sa distanța minimă calculată prin adăugarea la distanța curentă a valorii 1.

Procedeul se repetă cât timp coada nu este vidă și poziția în care se află cea de-a doua poartă nu a fost explorată. Rezultatul se va obține în matricea A, pe poziția celei de a doua porți.

Programul complet arată astfel :

```
#include <cstdio>
#include <queue>
using namespace std;

const int dx[5] = {0, -1, 0, 1, 0};
const int dy[5] = {0, 0, 1, 0, -1};
queue<pair<int, int> > C;
int a[178][178];
int ix, iy, ox, oy, x, y;
int n, m;
```

1. Pentru aprofundarea algoritmului lui Lee puteți utiliza software-ul educațional „Algoritmul lui Lee”, prezentat pe Arhiva educațională .campion (<http://campion.edu.ro/arhiva/>). O prezentare detaliată a structurilor stivă și coadă cu aplicații puteți consulta în capitolul 5 din volumul I al lucrării *Programarea în limbajul C/C++ pentru liceu*.

```

int main()
{
    int k, i, j;
    pair<int, int> xx, yy;
    FILE *f = fopen("alee.in", "r");
    FILE *g = fopen("alee.out", "w");
    fscanf(f, "%d%d", &n, &m);
    for (k = 1; k <= m; k++)
        { fscanf(f, "%d%d", &i, &j); a[i][j] = -1; }
    fscanf(f, "%d%d%d%d", &ix, &iy, &ox, &oy);
    fclose(f);
    for (i = 1; i <= n; i++) //bordare
        a[i][0]=a[i][n+1]=a[0][i]=a[n+1][i]=-1;
    a[ix][iy] = 1;           //prima dala
    C.push(make_pair(ix, iy)); //pun in coada
    while (!C.empty() && a[ox][oy]==0)
        { xx=C.front(); C.pop(); //extrag element din coada
            for (k = 1; k <= 4; k++)
                { yy.first = xx.first + dx[k];
                    yy.second = xx.second + dy[k];
                    if (a[yy.first][yy.second]==0) //e liber
                        { a[yy.first][yy.second] =
                            a[xx.first][xx.second] + 1;
                            C.push(make_pair(yy.first, yy.second));
                        }
                }
        }
    fprintf(g, "%d\n", a[ox][oy]); //scrive rezultat
    fclose(g);
    return 0;
}

```

Aplicația 3. Algoritmul Bellman-Ford optimizat

Algoritmul Bellman-Ford² determină drumurile de cost minim de la un vârf din graf la fiecare dintre celelalte vârfuri, identificând eventualele circuite de cost negativ. În forma lui „standard”, complexitatea algoritmului Bellman-Ford este $O(n^3)$, unde cu n am notat numărul de vârfuri din graf.

Implementarea algoritmului Bellman-Ford poate fi optimizată utilizând o coadă.

Reprezentarea informațiilor

Vom reprezenta graful prin liste de adiacență cu costuri. Mai exact pentru fiecare vârf x din graf reținem arcele cu extremitatea inițială x într-un vector de perechi de numere întregi ($pair<int, int>$); primul număr din pereche ($first$) reprezintă extremitatea finală a arcului, iar al doilea număr din pereche ($second$) reprezintă costul arcului respectiv.

2. O prezentare detaliată a algoritmului Bellman-Ford poate fi consultată în capitolul 1 din volumul al III-lea al lucrării *Programarea în limbajul C/C++ pentru liceu*.

```
| vector < pair<int,int> > G[NMAX] ;
```

Costurile drumurilor minime le vom reține în vectorul $dmin$ ($dmin[x] =$ costul drumului minim de la vârful x_0 la vârful x ; dacă nu există drum de la x_0 la x , $dmin[x] = INF$, unde INF este o constantă suficient de mare).

```
| int dmin[NMAX] ;
```

Vom utiliza o coadă C , în care inserăm vârfurile grafului, pe măsură ce calculăm costurile drumurilor minime.

```
| queue<int> C;
```

Pentru a detecta dacă există circuite de cost negativ, trebuie să contorizăm de câte ori a fost introdus în coadă fiecare vârf. Pentru aceasta vom utiliza un vector nr , $nr[x]$ reprezentând de câte ori a fost introdus în coadă vârful x . Dacă $nr[x] > n$ deducem că am identificat un circuit de cost negativ (variabila logică negativ reține acest lucru).

```
| int nr[NMAX] ;
```

Inițial: $dmin[x] = INF$, pentru orice $x \neq x_0$ și $dmin[x_0] = 0$ și inserăm în coada C vârful de start x_0 . Cât timp în coada C mai există vârfuri și nu am identificat un circuit de cost negativ:

- extragem un vârf x din C ;
- parcurgem lista de adiacență a lui x și încercăm să optimizăm costurile drumurilor vârfurilor y care sunt extremități finale ale arcelor care pleacă din x . Vârfurile y pentru care costul drumului de la x_0 la y a fost optimizat sunt introduse în coadă (și contorizăm acest lucru în $nr[y]$).

```
#include <cstdio>
#include <vector>
#include <queue>
#define INF 1000000000
#define NMAX 50001
using namespace std;
int n, x0;
vector < pair<int,int> > G[NMAX];
int dmin[NMAX];
int nr[NMAX];
bool negativ;
queue<int> C;

void citire();
void bellman_ford();
void afisare();

int main()
{ citire();
  bellman_ford();
  afisare();
```

```

return 0;
}

void citire()
{int i, m, x, y, c;
FILE * fin=fopen("bellmanford.in", "r");
fscanf(fin,"%d %d %d", &n, &m, &x0);
for (i=0; i<m; i++)
{ fscanf(fin,"%d %d %d", &x, &y, &c);
G[x].push_back(make_pair(y,c)); }
}

void bellman_ford()
{vector< pair<int,int> > ::iterator it;
int i, x;
for (i=1; i<=n; ++i) dmin[i]=INF;
dmin[x0]=0;
C.push(x0);
while (!C.empty()&& !negativ)
{x=C.front(); C.pop();
for (it = G[x].begin(); it != G[x].end(); ++it)
if (dmin[it->first] > dmin[x]+it->second)
{ dmin[it->first]=dmin[x]+it->second;
nr[ it->first ]++;
C.push(it->first);
if (nr[it->first]>n) negativ=true;
}
}
}

void afisare()
{FILE * fout=fopen("bellmanford.out", "w");
if (negativ)
fprintf(fout,"Circuit de cost negativ!\n");
else
{ for (int i=1; i<=n; i++)
if (i!=x0)
fprintf(fout,"%d ", dmin[i]==INF?dmin[i]:0);
fprintf(fout,"\n");
},
fclose(fout);
}
}

```

Observații

1. Dacă dorim să reconstituim drumurile, este necesar ca pentru fiecare vârf y să reținem $\text{prec}[y] = \text{vârf} \text{ care îl precedă pe } y \text{ pe drumul minim de la } x_0 \text{ la } y$; ($\text{vârf} x_0$ nu are precedent, deci $\text{prec}[x_0]=0$). Când costul drumului minim de la x_0 la un vârf y se schimbă, reținem în $\text{prec}[y]$ vârful x care reprezintă extremitatea inițială a arcului care a determinat optimizarea.

2. Complexitatea teoretică a algoritmului Bellman-Ford este de $O(n*m)$, însă în practică se comportă mult mai bine, devenind comparabil cu algoritmul lui Dijkstra.

6.2. Clasa adaptor stack (stiva)

Clasa adaptor stack este definită pentru a realiza operațiile specifice unei structuri de date de tip **LIFO** (Last In First Out). Într-o astfel de structură – denumită *stivă* – elementele sunt inserate și extrase la un singur capăt, denumit vârful stivei.

Pentru a utiliza această clasă trebuie să includem fișierul antet stack :

```
#include <stack>
```

Declarația clasei adaptor stack :

```
template <class T, class Container=deque<T> > class stack;
```

Observați că această clasă şablon are doi parametri :

- T – reprezintă tipul elementelor containerului ;
- Container – reprezintă clasa container care este utilizată pentru memorarea elementelor ; implicit aceasta este clasa deque.

Constructorii și destructorul

Pentru clasa adaptor stack sunt definiți constructorul implicit, constructorul de copiere, precum și destructorul care distrugе obiectul, eliberând memoria alocată.

Exemple

```
stack<int> S;
```

Creează o stivă denumită S cu 0 elemente de tip int.

```
stack<pair<int, int> > Sp;
```

Creează o stivă denumită Sp cu 0 elemente de tip pair<int, int>.

```
list<int> L(50, 0);
stack<int, list<int> > SL(lista);
```

Creează o listă L cu 50 de elemente având valoarea 0, apoi creează o stivă SL cu 50 de elemente de tip int, copiind elementele din lista creată. Operația de inițializare a stivei cu elementele din listă a fost posibilă deoarece clasa container suport pentru stiva SL este list.

Funcțiile membre referitoare la dimensiunea containerului

Sunt disponibile funcția `size()`, care returnează numărul de elemente din stivă, și funcția `empty()`, care returnează valoarea `true` în cazul în care stiva este vidă.

Accesori

Singurul element dintr-o stivă la care avem acces este cel din vârful stivei. Funcția membră `top()` returnează o referință către elementul din vârful stivei.

Funcțiile membre care modifică containerul

Într-o stivă sunt permise două operații :

- inserarea unui element la vârful stivei

```
| void push (const value_type& val);
```

Inserează un element cu valoarea `val` la vârful stivei curente.

- extragerea elementului din vârful stivei

```
| void pop();
```

Elimină și distrugе elementul din vârful stivei. Acстt element este ultimul care a fost inserat în stivă.

Ca și în cazul adaptorului `queue`, începând cu `C++ 11` sunt definite funcția membră `emplace()`, care creează și inserează un element la vârful stivei, și funcția membră `swap()`, care interschimbă stiva curentă cu stiva specificată ca parametru.

Operatorii relaționali

Pentru clasa `stack` sunt supraîncărcați toți operatorii relaționali, compararea realizându-se din punct de vedere lexicografic.

Începând cu `C++ 11` este definită și funcția independentă `swap()` care schimbă între ele conținutul celor două containere `stack` indicate ca parametri. Acestea trebuie să fie de același tip, chiar dacă dimensiunea lor diferă.

Aplicația 4. Reacții

Să considerăm o secvență de n substanțe chimice $s=s_1, s_2, \dots, s_n$. Substanțele sunt numerotate distinct de la 1 la n și fiecare substanță apare în secvență s o singură dată. Să considerăm o subsecvență $s_{ij}=s_i s_{i+1} \dots s_j$ și să notăm cu \min_{ij} și \max_{ij} cel mai mic, respectiv cel mai mare număr din subsecvență. Subsecvența respectivă constituie un interval dacă ea conține toate numerele naturale cuprinse între \min_{ij} și \max_{ij} .

Cu substanțele din secvența s se vor efectua diferite experimente. În timpul unui experiment pot reacționa două substanțe alăturate s_i și s_{i+1} doar dacă numerele lor de ordine sunt consecutive. În urma reacției se obține o nouă substanță, formată din substanțele care au reacționat, notată (s_i, s_{i+1}) . Mai mult, substanțele obținute pot reacționa dacă ele sunt alăturate, iar prin reunirea subsecvențelor de substanțe ce le compun se obține un interval. Experimentul este declarat reușit dacă în final, urmând regulile de mai sus, se obține o singură substanță formată din toate cele n substanțe din secvența s , aceasta fiind declarată stabilă.

De exemplu, pentru $n=6$ substanțe și secvența $s=6, 3, 2, 1, 4, 5$ se poate proceda astfel :

Etapa	Acțiune	Configurație
1	Configurația inițială	6 3 2 1 4 5
2	Reacționează substanță 2 cu substanță 1	6 3 (2, 1) 4 5
3	Reacționează substanță 4 cu substanță 5	6 3 (2, 1) (4, 5)
4	Reacționează substanță 3 cu substanță (2, 1)	6 (3, 2, 1) (4, 5)
5	Reacționează substanță (3, 2, 1) cu substanță (4, 5)	6 (3, 2, 1, 4, 5)
6	Reacționează substanță 6 cu substanță (3, 2, 1, 4, 5)	(6, 3, 2, 1, 4, 5)

Nu din orice secvență de substanțe se poate obține în urma reacțiilor o substanță finală stabilă.

Cerință

Scrieți un program care să determine pentru o secvență dată de substanțe dacă în urma reacțiilor ce se produc conform regulilor din enunț rezultă o substanță stabilă.

Date de intrare

Fișierul de intrare `reactii.in` conține pe prima linie numărul natural n , numărul de substanțe. Pe cea de-a doua linie se află un număr natural m , reprezentând numărul de secvențe de căte n substanțe din fișierul de intrare. Fiecare dintre următoarele m linii conține căte n numere naturale distincte, separate prin căte un spațiu, reprezentând o secvență de n substanțe.

Date de ieșire

Fișierul de ieșire `reactii.out` conține, pentru fiecare secvență de substanțe din fișierul de intrare, căte o linie, pe care este afișată valoarea 1, dacă pentru secvența respectivă se poate obține o substanță stabilă, sau valoarea 0, în caz contrar.

Restricții

- $2 \leq n \leq 15000$
- $1 \leq m \leq 20$
- La un moment dat pot reacționa doar două substanțe.

Exemplu

reactii.in	reactii.out	Explicații
6	1	stabilă
4	0	nestabilă
6 3 2 1 5	1	stabilă
4	1	stabilă
3 4 1 6 5		
2		
2 3 1 5 4		
6		
6 2 3 1 4		
5		

(Olimpiada Națională de Informatică, Galați, 2009)

Soluție

Se utilizează o stivă S, ale cărei elemente sunt subsecvențe $p, p+1, \dots, p+k$ ale mulțimii $\{1, 2, \dots, n\}$. De fapt, un element al stivei va avea doar două componente, elementul minim și respectiv elementul maxim din subsecvență. În general, pentru a adăuga o substanță s la stiva S, se verifică dacă substanța din vârful stivei S formează o nouă substanță împreună cu s, cu alte cuvinte dacă reuniunea celor două seturi de valori este o substanță (conform regulii). În caz afirmativ, se extrage substanța din vârful stivei S, se formează noua substanță, apoi se repetă pașii anteriori (verificare, extragere, formare substanță nouă). Dacă reuniunea a două seturi de valori (substanțe) nu formează o nouă substanță, atunci ultima substanță nou-obținută se pune în vârful stivei S.

Astfel, setul de substanțe este parcurs o singură dată; se consideră inițial fiecare element ca fiind o substanță separată și se adaugă fiecare dintre aceste substanțe la S, în modul descris mai sus.

Dacă, la terminarea parcurgerii setului de valori, stiva S conține un singur element (o singură substanță), adică valorile 1 și n, substanța finală este stabilă.

Programul complet arată astfel :

```
#include <iostream>
#include <stack>
using namespace std;
struct SUBST //substanța retinuta prin minim si maxim
{ int minim, maxim; };
stack<SUBST> S;
FILE *f, *g;
int n, m, x, Substanta;
SUBST SVarf, SNoua;

bool OK(SUBST R, SUBST Rx) //verifica daca se pot combina
{ return !(R.maxim+1!=Rx.minim&&Rx.maxim+1!=R.minim) }
```

```

void Reuniune(SUBST R, SUBST &Rx)//combina cele 2 substante
{
    Rx.minim = min(Rx.minim, R.minim);
    Rx.maxim = max(Rx.maxim, R.maxim);
}

int main()
{
    f = fopen("reactii.in", "r");
    g = fopen("reactii.out", "w");
    fscanf(f, "%d%d", &n, &m);
    for (; m; --m) //ia pe rand fiecare configuratie
    {
        while (!S.empty()) S.pop(); //GolesteStiva
        for (Substanta=1; Substanta<=n; ++Substanta)
        {
            fscanf(f, "%d", &x);
            SNoua.minim = x; SNoua.maxim = x;// retin
            if (!S.empty()) //daca am elemente in stiva
            {
                SVarf=S.top();
                while (OK(SVarf, SNoua)) //daca DA
                {
                    S.pop();
                    Reuniune(SVarf, SNoua);
                    if (!S.empty())
                        SVarf = S.top();
                    else
                        break;
                }
                S.push(SNoua);
            }
            else //este prima substanta
                S.push(SNoua); //o pun direct in stiva
        }
        fprintf(g,"%d\n", S.size()==1?1:0);
    }
    fclose(f); fclose(g);
    return 0;
}

```

Aplicația 5. Cetăți

Terraland este un regat care se extinde permanent. Frontiera regatului este un poligon convex, în vârfurile căruia sunt construite cetăți pentru apărarea segmentelor de frontieră adiacente. La fiecare extindere a regatului frontiera se modifică: se construiesc cetăți noi, iar unele cetăți, construite anterior își pierd rolul de puncte de apărare. În toate cetățile se află garnizoane. Pentru a micșora cheltuielile pentru întreținerea armatei, MegaBit, regele Terralandului, a decis să demobilizeze garnizoanele din cetățile care nu mai au rolul de puncte de apărare (nu se mai află pe frontiera regatului).

Cerință

Scrieți un program, care va determina din câte cetăți vor fi demobilizate garnizoanele.

Date de intrare

Prima linie a fișierului de intrare `cetati.in` conține un număr întreg N – numărul de cetăți din Terraland. Urmează N linii ce conțin câte două numere întregi x_i , y_i , separate prin spațiu – coordonatele cetăților.

Date de ieșire

Fișierul de ieșire `cetati.out` va conține o singură linie pe care va fi afișat numărul de cetăți din care pot fi demobilizate garnizoanele.

Restricții

- $3 \leq N \leq 10000$
- $-10^9 \leq x_i, y_i \leq 10^9$

Exemplu

<code>cetati.in</code>	<code>cetati.out</code>	<i>Explicație</i>
<pre>10 2 1 3 4 6 3 6 6 8 5 10 3 11 7 12 6 9 11 15 8</pre>	5	

(.campion 2009)

Soluție

Punctele care au rol de apărare sunt vârfurile înfășurătorii convexe. Pentru a determina înfășurătoarea convexă vom utiliza algoritmul lui *Graham*³.

Programul complet arată astfel :

```
#include <cstdio>
#include <algorithm>
#include <stack>
#include <vector>
#define Nmax 10010
using namespace std;
```

3. O prezentare detaliată a algoritmului lui Graham poate fi consultată în capitolul 6 din volumul al III-lea al lucrării *Programarea în limbajul C/C++ pentru liceu*.

```

struct punct { double x, y, tg; } aux;
int n, i, poz, p, p1, p2;
double xmin, ymin;
vector<punct> v(Nmax);
stack<int> S;

int arie(double x1, double y1, double x2, double y2, double
x3, double y3)
{ double aria = (x3-x1) * (y2-y1) - (x2-x1) * (y3-y1);
  return (aria >= 0); }

int cmp(punct a, punct b)
{ return a.tg < b.tg; }

int main()
{ FILE * f = fopen("cetati.in", "r");
  FILE * g = fopen("cetati.out", "w");
  fscanf(f, "%d", &n);
  fscanf(f, "%lf %lf", &v[1].x, &v[1].y);
  poz = 1; xmin = v[1].x; ymin = v[1].y;
  for (i = 2; i <= n; i++)
    { fscanf(f, "%lf %lf", &v[i].x, &v[i].y);
      if(v[i].x<xmin || (v[i].x==xmin && v[i].y<ymin))
        poz = i, xmin = v[i].x, ymin = v[i].y;
    }
  aux = v[1]; v[1] = v[poz]; v[poz] = aux;
  v[1].x = v[1].y = 0; //pun v[1] in origine
  v[n + 1] = v[1]; //inchei in acelasi punct
  S.push(0); //fictiv pe poz 0
  S.push(1); //1 pe prima pozitie
  for (i = 2; i <= n; i++) //pentru celelalte puncte
    { v[i].x-= xmin; v[i].y-= ymin; //translatez punctul
      if (v[i].x == 0) v[i].tg=1<<30;//tangenta infinit
       else v[i].tg = v[i].y/v[i].x; //tg calculata
    }
  sort(v.begin() + 2, v.begin() + 1 + n, cmp); //ordonez
  for (i = 2; i <= n + 1; i++) //pentru fiecare punct
    { S.push(i); //pun punctul in stiva
      p=S.top(); S.pop(); //consider ultimele 3 puncte
      p1=S.top(); S.pop();
      p2=S.top();
      while (S.size()>1 &&
arie(v[p1].x,v[p1].y,v[p].x,v[p].y,v[p2].x,v[p2].y))
        { S.push(p); //elimin punctul din mijloc
          p=S.top(); S.pop(); //consider iar ultimele 3
          p1=S.top(); S.pop(); //puncte
          p2=S.top();
        }
      S.push(p1); //repun pe stiva ultimele 2 puncte
    }
}

```

```

    S.push(p);
}
fprintf(g, "%d\n", n-(S.size()-2));
fclose(f); fclose(g);
return 0;
}

```

6.3. Clasa adaptor priority_queue

Pentru a utiliza această clasă trebuie să includem fișierul antet queue :

```
#include <queue>
```

Containerul adaptor priority_queue implementează o structură de date abstractă denumită coadă cu prioritate care permite executarea eficientă a operațiilor de inserare a unui element și de extragere a maximului. Elementele containerului sunt structurate ca un *max-heap*⁴.

```

template <class T, class Container=vector<T>,
          class Compare=less<typename Container::value_type> >
class priority_queue;
```

Din declarația clasei observați că există trei tipuri specificate ca parametru :

- T – reprezintă tipul elementelor structurii ;
- Container – reprezintă clasa container care este utilizată pentru memorarea elementelor ; implicit aceasta este clasa vector ; clasa container utilizată trebuie să disponă de iteratori cu acces aleator.
- Compare – reprezintă criteriul utilizat pentru compararea elementelor – o funcție sau un functor care compară două elemente și returnează un rezultat de tip bool ; implicit acesta este operatorul <.

Constructorii și destructorul

Constructorii clasei priority_queue sunt :

- Constructorul de inițializare :

```

explicit priority_queue (const Compare& comp = Compare(),
                        const Container& ctnr=Container());
```

- Constructorul bazat pe un domeniu

```

template <class InputIterator>
priority_queue (InputIterator prim, InputIterator ultim,
               const Compare& comp = Compare(),
               const Container& ctnr = Container());
```

4. Această structură de date este detaliat prezentată în capitolul 3 din volumul al III-lea al lucrării *Programarea în limbajul C/C++ pentru liceu*.

Constructorii inițializează criteriul de comparare, precum și un obiect container. Al doilea constructor inserează în container elementele din domeniul [prim, ultim), apoi apelează algoritmul `make_heap()` pentru a organiza elementele din container ca un *max-heap*.

Exemple

```
| priority_queue<int> pq1;
```

Creează o coadă cu prioritate vidă; elementele sunt de tip `int` și containerul care le stochează este cel implicit, adică `vector`; criteriul de comparare este cel implicit (operatorul `<`).

```
| priority_queue<int, vector<int>, greater<int> > pq2;
```

Creează o coadă cu prioritate vidă; elementele sunt de tip `int` și containerul care le stochează este cel implicit, adică `vector`; criteriul de comparare este functorul `greater` (care utilizează operatorul `>`); astfel se obține un *min-heap*.

```
| string s("Ana are mere");
| priority_queue<char> pq3 (s.begin(), s.begin+5);
```

Creează o coadă cu prioritate în care plasează primele 5 caractere din sirul `s` și le organizează ca *max-heap*; containerul care le stochează este cel implicit, adică `vector`; criteriul de comparare este cel implicit (operatorul `<`).

```
| struct Punct
| {
|     double x, y;
|     friend bool operator <(const Punct&, const Punct&);
| };
|
| bool operator <(const Punct& p1, const Punct& p2)
| { return p1.x<p2.x || p1.x==p2.x && p1.y<p2.y; }
```

Am definit un tip denumit `Punct` (reprezentând un punct prin coordonatele sale carteziene) și am supraîncărcat operatorul `<` pentru tipul `Punct`.

```
| priority_queue<Punct> pq4;
```

Creează o coadă cu prioritate vidă cu elemente de tip `Punct`; se utilizează containerul implicit (`vector`) și operatorul `<` pentru comparare.

```
| class ComparPunct
| {
|     public:
|         bool operator ()(const Punct& p1, const Punct& p2)
|         { return p1.x<p2.x || p1.x==p2.x && p1.y<p2.y; }
| };
```

O altă variantă de implementare ar fi să definim un functor de comparare a punctelor.

```
| priority_queue<Punct, deque<Punct>, ComparPunct> pq5;
```

Am creat o coadă cu prioritate cu elemente de tip `Punct`, containerul utilizat pentru stocarea elementelor este `deque`, iar compararea se realizează cu functorul `ComparPunct`.

Functiile membre publice

Clasa adaptor `priority_queue` are o interfață simplă, foarte asemănătoare cu interfața clasei `queue`:

| **bool** empty() **const**;

Returnează `true` dacă numărul de elemente este 0.

| **size_type** size() **const**;

Returnează numărul de elemente.

| **const value_type&** top() **const**;

Returnează o referință la primul element (cel cu prioritatea maximă).

| **void** push (**const value_type&** val);

Inserează valoarea `val`; funcția apelează `push_back()` din containerul suport, apoi apelează `push_heap()` pentru a restaura proprietatea de `heap`.

| **void** pop();

Extrage elementul cu prioritatea maximă; este apelată funcția `pop_heap()`, apoi `pop_back()` pentru containerul suport.

| **template <class... Args> void emplace (Args&&... args);**

Funcție specifică `C++ 11`, creează un element (apelând constructorul adecvat în funcție de parametrii specificați) și inserează elementul creat.

| **void** swap (`priority_queue`& x) **noexcept**;

Interschimbă coada cu prioritate curentă cu coada cu prioritate `x`, transmisă ca parametru (funcție disponibilă începând cu `C++ 11`). Începând cu `C++ 11`, este supraincărcată funcția `swap()` și ca funcție independentă (nemembră a clasei).

Observații finale

- Elementele dintr-un `priority_queue` fiind structurate ca un `heap`, operațiile `push()` și `pop()` se execută în timp logaritmic. Pentru `stack` și `queue` aceste operații se execută în timp constant.
- Niciuna dintre clasele adaptor `stack`, `queue` și `priority_queue` nu dispune de iteratori.
- Operațiile definite pentru clasele adaptor utilizează funcții ale clasei `container` suport. De exemplu, pentru toate cele 3 clase adaptor standard, clasa `container` suport trebuie să aibă definită funcția `push_back()` (utilizată de funcția `push()` ale claselor adaptor). Clasele `stack` și `priority_queue` utilizează pentru

funcția `pop()` funcția clasei suport `pop_back()`, în timp de clasa `queue` utilizează pentru funcția `pop()` funcția clasei suport `pop_front()`. În ceea ce privește accesul la elemente, pentru clasa `adaptor queue` sunt necesare funcțiile `front()` și `back()`, pentru clasa `stack` este necesară funcția `back()`, iar pentru `priority_queue` este necesară funcția `front()`.

4. Operatorii relaționali nu sunt supraîncărcați pentru `priority_queue`, dar sunt supraîncărcați pentru `stack` și `queue`, considerând ordinea lexicografică.

Aplicația 6. Piloți

Vasile are o companie de transport aerian. Pentru a se menține pe piață, el trebuie să reducă cheltuielile cât mai mult posibil.

La compania sa există N piloți (N par). Piloții sunt numerotați de la 1 la N în ordinea crescătoare a vîrstei (pilotul 1 este cel mai tânăr, pilotul N cel mai bătrân).

Vasile trebuie să formeze $N/2$ echipaže. Un echipaj este format din doi piloți (căpitanul și asistentul său). Căpitanul trebuie să fie mai în vîrstă decât asistentul său. În contractul fiecărui pilot sunt prevăzute două salarii: unul pentru cazul în care el este căpitan al echipajului, celălalt pentru cazul în care el este asistent. Evident, pentru orice pilot salariul său de căpitan este mai mare decât salariul său de asistent. Salariile pot să difere de la un pilot la altul. Chiar se poate întâmpla ca salariul căpitanului să fie mai mic decât salariul asistentului său.

Pentru a cheltui cât mai puțini bani pe salariile piloților, Vasile trebuie să determine o distribuire optimală a piloților pe echipaže.

Cerință

Scrieți un program care să determine suma minimă necesară pentru a plăti salariile piloților.

Date de intrare

Fișierul de intrare `piloti.in` conține pe prima linie un număr natural N reprezentând numărul de piloți. Pe următoarele N linii sunt informații despre salariile piloților. Pe linia $i+1$ se află două numere naturale c și a separate printr-un spațiu (c reprezintă salariul pilotului i pe post de căpitan, iar a reprezintă salariul pilotului i pe post de asistent).

Date de ieșire

Fișierul de ieșire `piloti.out` va conține o singură linie pe care va fi afișată suma minimă necesară pentru a plăti salariile celor N piloți.

Restricții

- $2 \leq N \leq 10000$, N par
- Pentru orice pilot, $1 \leq a < c \leq 100\ 000$

Exemple

piloti.in	piloti.out	piloti.in	piloti.out
6	32000	6	33000
10000 7000		5000 3000	
9000 3000		4000 1000	
6000 4000		9000 7000	
5000 1000		11000 5000	
9000 3000		7000 3000	
8000 6000		8000 6000	

(.campion 2004)

*Soluție*Metoda de rezolvare este *Greedy*.

Primul pilot trebuie să fie asistent (fiind cel mai Tânăr). Dintre piloții 2 și 3 unul trebuie să fie asistent. Dintre piloții 4, 5 și cel care a rămas dintre 2 și 3 unul trebuie să fie asistent. Deci la fiecare doi piloti citiți trebuie să aleg unul dintre cei existenți care să fie plasat pe post de asistent (alegerea se face dintre toți piloții citiți deja și care nu au fost stabiliți ca asistenți). Criteriul de alegere: diferența dintre salariul său de căpitan și salariul său de asistent să fie maximă.

Pentru a determina suma totală minimă necesară pentru a plăti salariile piloților, vom utiliza o variabilă *rez*. Când citesc un pilot, voi considera că el este căpitan și voi adăuga salariul lui de căpitan la *rez*, iar diferența dintre salariul său de căpitan și salariul său de asistent o voi insera într-un *max-heap* *H*. Dar după fiecare două citiri trebuie să aleg un asistent. În acest scop, voi extrage maximul din *max-heap-ul H*. Din *rez* voi scădea maximul extras (ceea ce înseamnă că pilotul corespunzător devine asistent)⁵.

```
#include <fstream>
#include <queue>
using namespace std;
ofstream fout("piloti.out");
ifstream fin("piloti.in");

priority_queue<int> H;

int main(void)
{int n, rez = 0, capitan, asistent, i;
fin>>n;
for (i=0; i<n; ++i)
{ fin >> capitan >> asistent;
rez += capitan;
H.push(capitan-asistent);
if (i%2==0)
{ rez -=H.top(); H.pop(); }}
```

5. O implementare fără *STL* poate fi consultată în capitolul 3 din volumul al III-lea al lucrării *Programarea în limbajul C/C++ pentru liceu*.

```

    }
fout << rez << '\n';
fout.close();
return 0;
}

```

Aplicația 7. Ksecv

Fie un sir a_1, a_2, \dots, a_N de N numere întregi și K un număr natural. O secvență $a_i, a_{i+1}, a_{i+2}, \dots, a_j$ din sir spunem că este K -secvență dacă există cel mult K numere strict mai mari decât a_i .

De exemplu, pentru $N=10$, sirul $a=3, 6, 4, 2, 1, 10, 9, 5, 8, 7$ și $K=3$, $6, 4, 2, 1, 10, 9, 5, 8$ este K -secvență pentru că sunt exact 3 numere strict mai mari decât 6. La fel, $5, 8, 7$ este K -secvență pentru că sunt doar două numere strict mai mari decât 5.

Cerință

Scrieți un program care să determine începutul și sfârșitul pentru cea mai lungă K -secvență din sir.

Date de intrare

Fișierul de intrare ksecv.in conține pe prima linie numerele naturale N și K , iar pe a doua linie elementele a_1, a_2, \dots, a_N ale sirului, separate prin câte un spațiu.

Date de ieșire

Fișierul de ieșire ksecv.out va conține pe prima linie două numere naturale incmax și sfmax separate printr-un spațiu, reprezentând indicii de început și de sfârșit ai K -secvenței de lungime maximă. Dacă sunt mai multe soluții, se va scrie aceea pentru care incmax este minim.

Restricții

- $3 \leq N \leq 100\ 000$
- $1 \leq K \leq N$
- $-10^9 \leq a_i \leq 10^9$, pentru orice i între 1 și N .

Exemplu

ksecv.in	ksecv.out	Explicații
10 3 3 6 4 2 1 10 9 5 8 7	2 9	Cea mai lungă 3-secvență este cea care începe la poziția 2 și se încheie la poziția 9, adică 6 4 2 1 10 9 5 8, în această secvență existând 3 numere strict mai mari decât 6.

Soluție

Vom parcurge sirul a element cu element și la fiecare pas încercăm să construim cea mai lungă K-secvență care începe cu a_i . Pentru aceasta trebuie să ținem evidența elementelor strict mai mari decât a_i . Când numărul acestor elemente este egal cu K (sau am terminat de parcurs sirul a) atunci am obținut o K-secvență maximală, o comparăm cu cea mai lungă K-secvență și o reținem dacă este cazul. Apoi încercăm să construim o altă K-secvență. Pentru a nu obține un algoritm pătratic, în construcția secvenței următoare ne vom baza pe secvența curentă. Să notăm cu x începutul K-secvenței curente și cu a_i elementul curent pe care îl analizăm din sirul a. Există două cazuri posibile :

- $x \geq a_i$. În acest caz nu are sens să construim K-secvența care începe cu a_i , deoarece lungimea acesteia nu va fi mai mare decât lungimea maximă curentă.
- $x < a_i$. În acest caz vom construi K-secvența care începe cu a_i . În K-secvența curentă noi aveam K elemente mai mari decât x. Dintre acestea unele sunt mai mari decât a_i , altele nu. Elementele care sunt mai mari decât a_i le contorizăm în secvența care va începe cu a_i , dar celelalte nu. Prin urmare, trebuie să eliminăm din structura de date în care ținem evidența elementelor $> x$ din secvența curentă pe cele mai mici (adică cele care sunt $\leq a_i$). Deducem că structura de date în care reținem evidența elementelor $> x$ trebuie să permită executarea eficientă a două operații : extragerea minimului și inserarea. Prin urmare trebuie să fie un *min-heap*. Revenind : extragem din *min-heap* cel mai mic element, cât timp acesta este $\leq a_i$. Apoi completăm secvența care începe cu a_i (adică adăugăm elemente la secvență atât timp cât *min-heap*-ul nu are lungimea $> K$). Aceasta va deveni secvența curentă, iar procedeul descris se repetă.

```
#include <iostream>
#include <queue>
#define NMAX 100000
using namespace std;
int n, k, lgmax, incmax, sfmax;
int a[NMAX];
priority_queue<int, vector<int>, greater<int> > H;

int main()
{int i, x, sf;
 FILE * fin=fopen("ksecv.in", "r");
 FILE *fout=fopen("ksecv.out", "w");
 fscanf(fin, "%d %d", &n, &k);
 for (i=0; i<n; i++) fscanf(fin, "%d", &a[i]);
 x=a[0]-1; sf=0;
 for (i=0; i<n; i++)
 if (a[i]>x) //construim o k-secventa incepand cu a[i]
 { //eliminam din heap elementele <= a[i]
 while (!H.empty() && H.top()<=a[i]) H.pop();
 //adaugam elemente cat timp e posibil
}
```

```

while (sf<n)
    if (a[sf]<=a[i]) sf++;
    else
        if (H.size()<k) {H.push(a[sf]); sf++; }
        else break;
    //compar lungimea secventei curente cu lgmax
    if (lgmax<(sf-i))
        {lgmax=sf-i; incmax=i+1; sfmax=sf; }
    x=a[i];
}
fprintf(fout,"%d %d\n", incmax, sfmax);
fclose(fout);
return 0;
}

```

Aplicația 8. Algoritmul lui Kruskal optimizat

Algoritmul lui Kruskal⁶ construiește un arbore parțial de cost minim pentru un graf neorientat conex și ponderat.

Metoda utilizată este *Greedy*: cât timp este posibil, selectăm o muchie de cost minim care nu a mai fost selectată și care nu formează cicluri cu muchiile deja selectate. Numărul de muchii selectate va fi $n-1$ (unde cu n am notat numărul de vârfuri din graf).

Pentru a selecta eficient o muchie de cost minim, vom organiza muchiile ca un *min-heap*, în funcție de cost.

Pentru a testa dacă o muchie formează cicluri cu muchiile deja selectate, trebuie să ținem evidență componentelor conexe formate cu muchiile deja selectate. Pentru o implementare eficientă vom utiliza algoritmii *Union-Find*⁷ optimizați.

Reprezentarea informațiilor

1. Vom defini un struct denumit **Muchie** în care reținem extremitățile unei muchii și costul acesteia. Supraîncărcăm operatorul $>$ pentru compararea muchiilor (deoarece dorim să organizăm muchiile ca un *min-heap*).
2. În coada cu prioritate **H** reținem muchiile grafului, organizate ca un *min-heap* în funcție de cost.
3. În vectorul **sol** vom reține muchiile ce formează arborele parțial de cost minim; în **costmin** reținem suma costurilor muchiilor selectate.
4. Pentru a reprezenta componentele conexe ale grafului (care sunt mulțimi disjuncte de muchii), vom considera că fiecare componentă conexă este un arbore. Reprezentăm arborii prin referințe ascendente (mai exact, pentru fiecare vârf x reținem **tata[x]** = nodul părinte al lui x , respectiv 0 dacă x este rădăcina arborelui). Operația **Find(x)** determină rădăcina arborelui din care face parte x (și, pentru
6. O descriere detaliată și o implementare pentru algoritmul lui Kruskal puteți consulta în capitolul 1 din volumul al III-lea al lucrării *Programarea în limbajul C/C++ pentru liceu*.
7. Reprezentarea mulțimilor disjuncte și implementarea optimizată a algoritmilor *Union-Find* se află în capitolul 3 din volumul al III-lea al lucrării *Programarea în limbajul C/C++ pentru liceu*.

eficiență, compresează drumul de la x la rădăcină). Operația Union($c1, c2$) unifică arborii corespunzători componentelor conexe $c1$ și $c2$ (rădăcina arborelui cu înălțimea mai mică devine fiul rădăcinii celuilalt). Pentru a implementa operația Union eficient este necesar să reținem și înălțimea fiecărui arbore format ($h[x] =$ înălțimea arborelui cu rădăcina x).

```
#include <fstream>
#include <queue>
#define NMAX 1001
using namespace std;

struct Muchie
{
    int x, y;
    double cost;
    friend bool operator >(const Muchie&, const Muchie&);
};

bool operator >(const Muchie& m1, const Muchie& m2)
{ return m1.cost > m2.cost; }

priority_queue<Muchie, vector<Muchie>, greater<Muchie> > H;
vector<Muchie> sol;
int tata[NMAX];
int h[NMAX];
int n;
double costmin;

void Citire();
void Afisare();
int Find (int);
void Union(int, int);

int main()
{int c1, c2;
Muchie m;
Citire();
while (sol.size() < n-1)
    //cat timp nu am selectat cele n-1 muchii
    {//extrag din H muchia de cost minim
m=H.top(); H.pop();
//determin componentele conexe din care fac parte
//extremitatile muchiei selectate m
c1=Find(m.x); c2=Find(m.y);
if (c1!=c2)
    //m nu formeaza cicluri cu muchiile selectate
    {//selectez aceasta muchie
    costmin+=m.cost; sol.push_back(m);
    //unific componentele conexe ale extremitatilor
    Union(c1, c2); }
}
}
```

```

Afisare();
return 0;
}

void Citire()
{Muchie m;
int i, nrm;
FILE * fin=fopen("kruskal.in", "r");
fscanf(fin,"%d %d", &n, &nrm);
for (i=0; i<nrm; i++)
{fscanf(fin,"%d %d %lf", &m.x, &m.y, &m.cost);
H.push(m); }
}

void Afisare()
{FILE * fout=fopen("kruskal.out", "w");
fprintf(fout,"%.2lf\n", costmin);
for (int i=0; i<n-1; i++)
fprintf(fout,"%d %d\n",sol[i].x,sol[i].y);
fclose(fout);
}

int Find (int x)
{int r=x;
//determinam radacina arborelui
while (tata[r]) r=tata[r];
//compresam drumul
int y=x, aux;
while (y!=r)
{ aux=tata[y]; tata[y]=r; y=aux; }
return r;
}

void Union (int c1, int c2)
{
if (h[c1]>h[c2]) tata[c2]=c1;
else
{tata[c1]=c2;
if (h[c1]==h[c2]) h[c2]++;
}
}

```

Aplicația 9. Algoritmul lui Dijkstra optimizat

Algoritmul lui Dijkstra⁸ determină un drum de cost minim de la vârful de start x_0 la fiecare dintre celealte vârfuri ale unui graf ponderat dat.

8. O descriere detaliată și o implementare pentru algoritmul lui Dijkstra puteți consulta în capitolul 1 din volumul al III-lea al lucrării *Programarea în limbajul C/C++ pentru liceu*.

Reprezentarea informațiilor

Vom reprezenta graful prin liste de adiacență cu costuri. Mai exact pentru fiecare vârf x din graf reținem arcele cu extremitatea inițială x într-un vector de perechi de numere întregi ($\text{pair}<\text{int}, \text{int}>$); primul număr din pereche (first) reprezintă extremitatea finală a arcului, iar al doilea număr din pereche (second) reprezintă costul arcului respectiv.

```
| vector < pair<int, int> > G[NMAX];
```

Costurile drumurilor minime le vom reține în vectorul $dmin$ ($dmin[x] =$ costul drumului minim de la vârful x_0 la vârful x ; dacă nu există drum de la x_0 la x , $dmin[x] = INF$, unde INF este o constantă suficient de mare).

```
| int dmin[NMAX];
```

Pentru a reconstitui drumurile, este necesar ca pentru fiecare vârf x să reținem $\text{prec}[x] =$ vârful care îl precedă pe x pe drumul minim de la x_0 la x ; (vârful x_0 nu are precedent, deci $\text{prec}[x_0] = 0$).

```
| int prec[NMAX];
```

Vom utiliza o coadă cu prioritate H , în care inserăm vâfurile grafului pe măsură ce acestea sunt parcurse, astfel încât la fiecare pas să putem extrage vârful pentru care $dmin$ este minim (*min-heap*).

```
| priority_queue<int, vector<int>, ComparVF> H;
```

ComparVF este functorul care compară două vârfuri :

```
class ComparVF
{
public:
    bool operator () (const int& x, const int& y)
    { return dmin[x] > dmin[y]; }
};
```

Înțeles :

- $dmin[x] = INF$, pentru orice $x \neq x_0$ și $dmin[x_0] = 0$;
- $\text{prec}[x] = x_0$, pentru orice $x \neq x_0$ și $\text{prec}[x_0] = 0$;
- inserăm în coada cu prioritate H vârful de start x_0 .

Cât timp în coada cu prioritate H mai există vârfuri :

- extragem un vârf vf_{\min} din H (vârful situat la distanță minimă de x_0);
- parcurgem lista de adiacență a vârfului vf_{\min} și încercăm să optimizăm costurile drumurilor vârfurilor care sunt extremități finale ale arcelor care pleacă din vf_{\min} .

```
#include <cstdio>
#include <vector>
#include <queue>
#define INF 1000000000
#define NMAX 50001
```

```
using namespace std;

int n, x0;
vector < pair<int,int> > G[NMAX];
int dmin[NMAX];
int prec[NMAX];
FILE * fout;

class ComparVf
{public:
    bool operator () (const int& x, const int& y)
    { return dmin[x]>dmin[y]; }
};

priority_queue<int, vector<int>, ComparVf> H;

void citire();
void dijkstra();
void afisare();

int main()
{
    citire();
    dijkstra();
    afisare();
    return 0;
}

void citire()
{
    int i, m, x, y, c;
    FILE * fin=fopen("dijkstra.in","r");
    fscanf(fin,"%d %d %d", &n, &m, &x0);
    for (i=0; i<m; i++)
        { fscanf(fin,"%d %d %d", &x, &y, &c);
          G[x].push_back(make_pair(y,c));
        }
}

void dijkstra()
{
    int i, vfmin;
    for (i=1; i<=n; i++) {dmin[i]=INF; prec[i]=x0;}
    dmin[x0]=0; prec[x0]=0;
    H.push(x0);
    while (!H.empty())
        {vfmin=H.top(); H.pop();
         for (i=0; i<G[vfmin].size(); i++)
             //parcure lista de adiacenta a lui vfmin
             if (dmin[G[vfmin][i].first] >
                 dmin[vfmin]+G[vfmin][i].second)//optimizez
                 { dmin[G[vfmin][i].first] =
                     dmin[vfmin]+G[vfmin][i].second;

```

```

        prec[G[vfmin][i].first]=vfmin;
        H.push(G[vfmin][i].first);
    }
}

void drum (int x)
{if (x)
{drum(prec[x]);
fprintf(fout,"%d ", x); }
}

void afisare()
{fout=fopen("dijkstra.out", "w");
for (int i=1; i<=n; i++)
if (i!=x0)
if (dmin[i]==INF)
fprintf(fout,"0\n");
else
{
fprintf(fout,"%d: ", dmin[i]);
drum(i);
fprintf(fout,"\n");
}
fclose(fout);
}
}

```

Observații

1. Același vârf poate fi inserat în coada cu prioritate H de mai multe ori. Ca urmare, implementând astfel algoritmul lui Dijkstra, dimensiunea memoriei necesare pentru coada cu prioritate H este de $O(m)$, unde cu m am notat numărul de arce din graf. Memoria necesară pentru H ar putea fi de $O(n)$ (unde cu n am notat numărul de vârfuri din graf), doar dacă atunci când optimizăm costul drumului către un vârf y nu îl inserăm în H , ci verificăm dacă el există sau nu în H . Dacă y nu există în H , îl inserăm. Dacă y există deja în H , nu îl inserăm, ci îl promovăm în $heap$ pe y , conform noii sale valori. Pentru aceasta ar trebui să implementăm noi operațiile cu $heap$ -ul pentru a putea reține și poziția fiecărui element în $heap$.
2. Complexitatea algoritmului lui Dijkstra implementat cu $heap$ este $O(m \log n)$.

6.4. Exerciții și probleme propuse

1. Să considerăm următoarea secvență de operații :

```

C.push(7);C.push(10);C.push(2);C.push(13);C.push(25);
while (!C.empty())
{ cout<<C.top()<<' ';
  C.pop(); }

```

Ce se va afișa pe ecran în cazul în care declarația obiectului C este :

- a. `stack<int> C;`
- b. `priority_queue<int> C;`

2. Scrieți o secvență formată dintr-un număr minim de operații `push()`, `pop()` și afișarea elementului de la vârf (`top()`) după execuția căreia, plecând de la o stivă inițial vidă, obținem pe ecran cuvântul mama.

3. Care dintre următoarele secvențe afișează elementele din coada cu prioritate H ?

| `priority_queue<int>H;`

a.

| `for (int i=0; i<H.size(); i++) cout<<H[i]<<'` ';

b.

| `priority_queue<int>::iterator it;`
| `for (it=H.begin(); it!=H.end(); ++it) cout<<*it<<'` ';

c.

| `while (!H.empty()) cout<<H.pop()<<'` ';

d.

| `while (!H.empty()) {cout<<H.top()<<'` ' ; H.pop();}

4. Care dintre următoarele declarații pentru obiectul H sunt corecte ?

a.

| `priority_queue<int, list<int> > H;`

b.

| `string s("mama");`
| `stack<char> H(s);`

c.

| `list<string> L(5, "mama");`
| `queue<string, list<string> > H(L);`

d.

| `priority_queue<int, deque<int>, greater > H;`

5. Care dintre următoarele afirmații sunt adevărate ?

- a. Orice clasă poate fi utilizată drept clasă suport pentru containerul adaptor `queue`.
- b. Operatorul de indexare nu este suprăîncărcat pentru clasa adaptor `stack`.
- c. La declararea unui obiect de tip `queue` trebuie să specificăm tipul elementelor, containerul suport, precum și criteriul de comparare a elementelor.
- d. Pentru toate containerele adaptor standard din *STL* inserarea și extragerea elementelor din container se realizează în timp constant.
- e. Niciuna dintre afirmațiile precedente.

6. Implementați parcurgerea în lățime (*BFS*) a unui graf, utilizând clasa `queue`.
7. Implementați algoritmul lui Prim de determinare a unui arbore parțial de cost minim, optimizându-l prin utilizarea unui *heap*. Pentru implementare, utilizați clasa adaptor `priority_queue`.
8. Implementați algoritmul Ford-Fulkerson de determinare a unui flux maxim într-o rețea de transport, utilizând biblioteca *STL*.
9. *Swap*

O parantezare corectă se obține aplicând una dintre următoarele reguli :

- sirul vid este o parantezare corectă ;
- dacă S este o parantezare corectă, atunci (S) este o parantezare corectă, iar cele două paranteze (și) care încadrează sirul S sunt denumite *paranteze pereche* ;
- dacă S_1, S_2, \dots, S_k sunt parantezări corecte atunci sirul $S_1S_2\dots S_k$ obținut prin concatenarea acestora este o parantezare corectă.

De exemplu sirurile $()$, $(())$, $(())$ și $(((())))$ reprezintă parantezări corecte, în timp ce $)()$, $(())$ și $(((()))$ nu sunt parantezări corecte.

Fie S un sir care reprezintă o parantezare corectă. Pentru fiecare dintre parantezele pereche din sirul S asociem un *cost* egal cu diferența dintre poziția pe care se află în S paranteza închisă și poziția parantezei deschise pereche. Pozițiile în sir le considerăm numerotate începând cu 1. *Costul total* al unei parantezări corecte îl reprezintă suma costurilor tuturor parantezelor pereche din aceasta.

De exemplu, sirul $(())$ este format din trei paranteze pereche, situate pe pozițiile 2 și 3, 4 și 5 și respectiv 1 și 6. Costul total al parantezării este $3 - 2 + 5 - 4 + 6 - 1 = 7$.

Numim operație *swap* interschimbarea a două paranteze situate în sir pe poziții alăturate. Această operație este *validă* doar dacă sirul nou obținut este la rândul său o parantezare corectă și dacă noua parantezare are costul total strict mai mic decât cea inițială.

Cerință

Scrieți un program care citește o succesiune de N caractere reprezentând o parantezare corectă și determină :

- Costul total asociat parantezării citite ;
- Costul minim al unei parantezări obținute prin efectuarea unei singure operații *swap* valide asupra parantezării citite ;
- Numărul de posibilități de a efectua o singură operație *swap* validă asupra parantezării inițiale pentru a obține costul determinat conform cerinței b).

Date de intrare

Fișierul de intrare `swap.in` conține pe prima linie numărul natural N și pe a doua linie o succesiune de N caractere reprezentând o parantezare corectă.

Date de ieșire

Fișierul de ieșire swap.out va conține pe prima linie un număr natural reprezentând costul parantezării citite. A doua linie va conține un număr natural reprezentând costul minim determinat conform cerinței b) sau valoarea -1 când nu se poate efectua nici o operație swap validă asupra parantezării citite. A treia linie a fișierului va conține un număr natural reprezentând răspunsul la cerința c) sau o dacă numărul afișat conform cerinței b) a fost -1.

Restricție

- $2 \leq N \leq 90000$

Exemplu

swap.in	swap.out	Explicații
8 () () ()	6 4 1	Pentru cerința a) costul parantezării este $2-1+6-3+5-4+8-7=6$. Executând o operație swap între parantezele de pe pozițiile 4 și 5 se obține sirul () () () care are costul 4, aceasta fiind singura posibilitate de a obține acest cost.

Olimpiada Națională de Informatică pentru Gimnaziu, 2013

10. Valet

Vasile lucrează într-o parcare. Sarcina lui este să scoată mașinile clienților din parcare. Terenul de parcare este dreptunghiular, format din $n \times m$ zone identice, aranjate pe n linii și m coloane. Linile sunt numerotate de la 1 la n , iar coloanele de la 1 la m . Ieșirea din parcare este în colțul parcării și are coordonatele $(1, 1)$. În fiecare zonă se poate afla o mașină, un stâlp sau poate fi liberă. Parcarea este plină, singura zonă liberă fiind ieșirea din parcare. Prin urmare, este foarte complicat să scoți o mașină din parcare. Pentru a-și face loc, Vasile poate muta o mașină din zona în care este plasată în una dintre zonele învecinate, dacă aceasta este liberă. Evident, stâlpii nu pot fi mutați, doar mașinile. Două zone sunt învecinate dacă se află pe aceeași linie, pe coloane consecutive, sau pe aceeași coloană, pe linii consecutive.

Cerință

Scrieți un program care să determine numărul minim de mutări de mașini pe care trebuie să le execute Vasile pentru a scoate o mașină din parcare, dacă acest lucru este posibil.

Date de intrare

Fișierul de intrare valet.in conține pe prima linie numerele naturale n și m separate prin spațiu. Urmează n linii, fiecare conținând câte m caractere din mulțimea $\{'.', '#', 'c', 'x'\}$. Caracterul '.' indică zona liberă. Caracterul '#' indică un stâlp. Caracterul 'c' indică o mașină parcată. Caracterul 'x' indică mașina pe care Vasile trebuie să o scoată din parcare. În fișierul de intrare există un singur caracter 'x' și un singur caracter '.', plasat în colțul $(1, 1)$ al parcării.

Date de ieșire

Fișierul de ieșire `valet.out` va conține o singură linie pe care va fi scris numărul minim de mutări pe care trebuie să le efectueze Vasile pentru a scoate mașina din parcare, dacă acest lucru este posibil. În caz contrar, pe prima linie se va scrie cuvântul **imposibil**.

Restricție

- $1 \leq n, m \leq 50$

Exemple

valet.in	valet.out	valet.in	valet.out
3 3 .#X ccc c#c	imposibil	2 3 .cX ccc	7

.campion 2011

11. Predecesor

Considerăm un sir de N numere naturale distințe a_1, a_2, \dots, a_N . Pentru fiecare termen a_i definim predecesorul său, dacă există, ca fiind cel mai din dreapta termen a_j , cu $j < i$ și $a_j < a_i$. De exemplu, pentru sirul 8, 12, 2, 4, 3, 10, 9, 7, 5, 6, numărul 8 este predecesorul lui 12, 3 este predecesorul lui 5, în timp ce 10, 4 și 2 nu sunt predecesorii niciunui număr.

Cerință

Scrieți un program care determină câte numere din sir nu sunt predecesori ai niciunui alt număr.

Date de intrare

Fișierul de intrare `predecesor.in` conține pe prima linie numărul natural N reprezentând numărul de termeni ai sirului. Pe următoarea linie se găsesc, separați prin câte un spațiu, termenii a_1, a_2, \dots, a_N ai sirului.

Date de ieșire

Fișierul de ieșire `predecesor.out` va conține o singura linie pe care va fi scris un singur număr natural reprezentând numărul de termeni ai sirului care nu sunt predecesori ai niciunui alt număr.

Restricții

- $3 \leq N \leq 500\ 000$
- $1 \leq a_i \leq 1\ 000\ 000\ 000$, pentru orice $1 \leq i \leq N$

Exemplu

predecesor.in	predecesor.out
10	6
8 12 2 4 3 10 9 7 5 6	

.campion 2009

12. Cezar

În Roma antică există n case senatoriale distințe, câte una pentru fiecare dintre cei n senatori ai Republicii. Casele senatoriale sunt numerotate de la 1 la n, între oricare două case existând legături directe sau indirecte. O legătură este directă dacă ea nu mai trece prin alte case senatoriale intermediare. Edilii au pavat unele dintre legăturile directe dintre două case (numind o astfel de legătură pavată „stradă”), astfel încât între oricare două case senatoriale să existe o singură succesiune de străzi prin care se poate ajunge de la o casă senatorială la cealaltă.

Toți senatorii trebuie să participe la ședințele Senatului. În acest scop, ei se deplasează cu lectica. Orice senator care se deplasează pe o stradă plătește 1 ban pentru că a fost transportat cu lectica pe acea stradă.

La alegerea sa ca prim consul, Cezar a promis că va dota Roma cu o lectică gratuită care să circule pe un număr de k străzi ale Romei astfel încât orice senator care va circula pe străzile respective să poată folosi lectica gratuită fără a plăti. Străzile pe care se deplasează lectica gratuită trebuie să fie legate între ele (zborul, metroul sau teleportarea nefiind posibile la acea vreme).

În plus, Cezar a promis să stabilească sediul sălii de ședințe a Senatului într-una dintre casele senatoriale aflate pe traseul lecticii gratuite. Problema este de a alege cele k străzi și amplasarea sediului sălii de ședințe a Senatului astfel încât, prin folosirea transportului gratuit, senatorii, în drumul lor spre sala de ședințe, să facă economii cât mai însemnante. În calculul costului total de transport, pentru toți senatorii, Cezar a considerat că fiecare senator va călători exact o dată de la casa sa până la sala de ședință a Senatului.

Cerință

Scrieți un program care determină costul minim care se poate obține prin alegerea adecvată a celor k străzi pe care va circula lectica gratuită și a locului de amplasare a sălii de ședință a Senatului.

Date de intrare

Fișierul *cezar.in* conține pe prima linie două valori n k separate printr-un spațiu reprezentând numărul total de senatori și numărul de străzi pe care circulă lectica gratuită. Pe următoarele n-1 linii se află câte două valori i și j separate printr-un spațiu, cu semnificația că între casele i și j există stradă.

Date de ieșire

Pe prima linie a fișierului cezar.out se va scrie costul total minim al transportării tuturor senatorilor pentru o alegere optimă a celor k străzi pe care va circula lectica gratuită și a locului unde va fi amplasată sala de ședințe a Senatului.

Restricții

- $1 < n \leq 10000$, $0 < k < n$
- $1 \leq i, j \leq n$, $i \neq j$
- Străzile din fișierul de intrare reprezintă două străzi distincte.

Exemple

cezar.in	cezar.out	Explicații
13 3 1 2 2 3 2 8 7 8 7 5 5 4 5 6 8 9 8 10 10 11 10 12 10 13	11	<p>Costul minim se obține, de exemplu, pentru alegerea celor 3 străzi între casele 5-7, 7-8, 8-10 și a sălii de ședințe a Senatului în casa 8:</p> <p>Există și alte alegeri pentru care se obține soluția 11.</p>

Olimpiada Județeană de Informatică 2007

7. Containere asociative

În biblioteca *STL* sunt definite patru containere asociative sortate: *set*, *multiset*, *map* și *multimap*. Începând cu *C++ 11*, biblioteca *STL* include și containere asociative nesortate: *unordered_set*, *unordered_multiset*, *unordered_map* și *unordered_multimap*.

În cadrul containerelor asociative elementele nu sunt identificate prin poziția acestora în cadrul containerului, ci prin intermediul unei *chei*. În cazul containerelor asociative sortate, elementele sunt stocate în ordinea cheilor, conform unui criteriu de comparare specificat. Pentru containerele asociative, operațiile de inserare, ștergere și căutare se realizează foarte eficient.

7.1. Containerele asociative sortate *set* și *multiset*

Containerul *set* stochează elemente distincte într-o anumită ordine. Vom denumi un container de tip *set* multime. Elementele într-o multime sunt identificate prin valoarea lor. Cu alte cuvinte, în cazul containerului asociativ *set*, cheia și valoarea coincid.

Containerul *multiset* este asemănător, diferența constând în faptul că valorile nu sunt distincte (ele se pot repeta). Vom denumi un container de tip *multiset* multimultime.

Pentru a utiliza aceste clase trebuie să includem fișierul antet *set*:

```
#include <set>

Declarația clasei set:

template < class T,
            class Compare = less<T>,
            class Alloc = allocator<T> >
> class set;
```

Clasa şablon *set* are trei tipuri specificate ca parametri:

- *T* – tipul valorilor stocate;
- *Compare* – criteriul utilizat pentru compararea elementelor (implicit se utilizează operatorul *<*);
- *Alloc* – tipul obiectului utilizat pentru alocarea memoriei pentru elementele containerului; implicit acesta este *allocator<T>*.

Declarația este similară și pentru multiset.

Criteriul de sortare specificat trebuie să fie antisimetric (dacă `Compare(x, y)` este `true`, atunci `Compare(y, x)` este `false`), tranzitiv (dacă `Compare(x, y)` este `true` și `Compare(y, z)` este `true`, atunci `Compare(x, z)` este `true`) și ireflexiv (`Compare(x, x)` este `false`). Același criteriu de sortare este utilizat și pentru a verifica dacă elementele sunt distincte (`x` și `y` sunt considerate egale dacă atât `Compare(x, y)`, cât și `Compare(y, x)` au valoarea `false`).

Clasele `set` și `multiset` nu permit modificarea elementelor stocate în container, însă permit inserarea unui element și extragerea unui element.

Iteratorii containerelor `set/multiset` sunt bidirecționali.

Deși standardul *STL* nu specifică explicit acest lucru, elementele containerelor `set/multiset` sunt organizate intern ca un arbore binar de căutare echilibrat (mai exact, un arbore roșu-și-negru). Acest lucru permite ca operațiile de căutare, inserare și extragere să se realizeze în timp logaritmic.

Constructorii și destructorul

Pentru clasele `set/multiset` sunt definiți constructorul implicit (care creează un container vid), constructorul bazat pe un domeniu (care creează un container în care copiază elementele dintr-un domeniu specificat), constructorul de copiere și destructorul, care distrug obiectul, eliberând memoria alocată dinamic. Începând cu `C++ 11`, sunt definiți constructorul de inițializare (care creează un container în care plasează elementele specificate într-o listă de inițializare) și constructorul de mutare (care creează un container în care mută elementele dintr-un container specificat).

Exemple

```
| set<int> C;
```

Creează o mulțime (`set`) denumită `C` cu 0 elemente de tip `int`. Ordinea elementelor este cea implicită, definită de operatorul `<`.

```
| string s("mama");
| set<char> d(s.begin(), s.end());
```

Creează un container de tip `set` în care plasează caracterele distincte din sirul `s`, ordinea caracterelor fiind cea implicită (`<`). Conținutul containerului `s` este `am`.

```
| multiset<char> d(s.begin(), s.end());
```

Creează un container de tip `multiset` în care plasează toate caracterele din sirul `s`, ordinea caracterelor fiind cea implicită (`<`). Conținutul containerului `ms` este `aamm`.

Să considerăm clasa `CompInt`, care permite compararea a doi întregi considerând doar ultima cifră :

```
class CompInt
{public:
    bool operator() (const int& x, const int& y)
    { return x%10 < y%10; }
};
```

Vom declara și vom popula un vector a de numere de tip int :

```
vector<int> a;
a.push_back(12); a.push_back(102); a.push_back(56);
a.push_back(213); a.push_back(212); a.push_back(14);
a.push_back(15); a.push_back(56);
```

Vectorul a conține acum valorile 12, 102, 56, 213, 212, 14, 15, 56.

Vom crea două mulțimi utilizând constructorul bazat pe un domeniu :

```
set<int> c1(a.begin(), a.end());
set<int, CompInt> c2(a.begin(), a.end());
```

În c1 vor fi plasate elementele distincte din vectorul a, în ordine crescătoare, considerând ordinea definită de operatorul <: 12 14 15 56 102 212 213. În c2 vor fi plasate elementele din vectorul a, elemente distincte luând în considerare ultima cifră, în ordinea crescătoare descrisă de CompInt : 12 213 14 15 56.

```
multiset<int, CompInt> c3(a.begin(), a.end());
```

În c3 sunt plasate toate elementele din vectorul a, în ordine crescătoare, luând în considerare doar ultima cifră: 12 102 212 213 14 15 56 56.

Funcțiile membre referitoare la dimensiunea containerului

Pentru clasele set/multiset sunt definite funcțiile empty(), size() și max_size(), cu semnificația uzuală.

Accesori

Containerele set/multiset conțin funcțiile membre begin(), end(), rbegin(), rend() care returnează iteratori cu semnificația uzuală. Începând cu C++ 11, sunt definite și funcțiile cbegin(), cend(), crbegin(), crend(), care returnează const_iterator. Reamintim că elementele acestor containere sunt constante (nu pot fi modificate), deci iteratorii indică elemente constante.

De asemenea, rețineți că elementele containerelor set/multiset sunt ordonate (conform criteriului de sortare specificat la declarare). Prin urmare, primul element al containerului este primul conform ordinii specificate („cel mai mic”), iar ultimul element este ultimul conform ordinii specificate („cel mai mare”).

Funcțiile membre care modifică containerul

- Inserare

```
pair<iterator,bool> insert (const value_type& val);
```

Inserează un element cu valoarea val. Pentru containerul set funcția returnează o pereche de valori, prima valoare din pereche (first) fiind un iterator care indică elementul nou inserat (sau poziția celui echivalent cu acesta existent în set); a doua valoare din pereche (second) este true dacă inserarea s-a realizat cu succes, respectiv false dacă inserarea nu a fost efectuată deoarece o valoare echivalentă

există deja în container. În cazul în care containerul este de tip multiset, inserarea se realizează întotdeauna, prin urmare funcția nu returnează o pereche de valori, ci doar iteratorul care indică poziția elementului nou inserat.

```
| iterator insert (iterator poz, const value_type& val);
```

Inserează un element cu valoarea val, începând căutarea poziției de inserare cu poziția indicată de iteratorul poz. Poziția poz reprezintă doar o indicație pentru funcția insert, ca poziție de început a căutării, și dacă poziția pe care trebuie inserată valoarea val este după poz, atunci inserarea este optimizată.

```
| template <class InputIterator>
| void insert (InputIterator prim, InputIterator ultim);
```

Inserează în containerul set/multiset elementele din domeniul [prim,ultim].

```
| template <class... Args>
| pair<iterator,bool> emplace (Args&&... p);
```

Disponibilă din C++ 11, funcția emplace() creează și apoi inserează un element în container.

```
| template <class... Args>
| iterator emplace_hint(const_iterator poz, Args&&... p);
```

Disponibilă din C++ 11, funcția emplace_hint() creează și apoi inserează un element în container, căutarea poziției de inserare începând cu poziția indicată de iteratorul poz.

- Extragere

```
| void erase (iterator poz);
```

Extrage din set/multiset elementul indicat de iteratorul poz.

```
| size_type erase (const value_type& val);
```

Caută în set/multiset valoarea val și extrage din set/multiset toate elementele care au această valoare. Funcția returnează numărul de elemente eliminate (pentru containerul set 1 sau 0, după cum valoarea a fost sau nu găsită).

```
| void erase (iterator prim, iterator ultim);
```

Elimină din set/multiset toate elementele din domeniul [prim,ultim].

```
| void clear();
```

Elimină toate elementele din container.

- Interschimbare

```
| void swap (set& x);
| void swap (multiset& x);
```

Funcție membră care interschimbă containerul curent cu containerul x transmis ca parametru.

Operațiile de căutare

| iterator find (**const** value_type& val) **const**;

Caută în set/multiset valoarea val și returnează un iterator care indică o poziție pe care se află această valoare. Dacă valoarea val nu a fost găsită, funcția returnează iteratorul end().

| size_type count (**const** value_type& val) **const**;

Returnează numărul de elemente din container echivalente (din perspectiva criteriului de sortare) cu valoarea val. Pentru set, această funcție poate returna doar 0 sau 1, deoarece valorile sunt distincte.

| iterator lower_bound (**const** value_type& val) **const**;

Returnează un iterator care indică poziția primului element din container care nu îl precedă pe val (acest element conține o valoare echivalentă cu val sau mai mare decât val, considerând criteriul de sortare specificat la declarare). Dacă toate elementele din container precedă valoarea val, atunci funcția returnează iteratorul end().

| iterator upper_bound (**const** value_type& val) **const**;

Returnează un iterator care indică poziția primului element din container care îl succedă pe val (acest element conține o valoare mai mare decât val, considerând criteriul de sortare specificat la declarare). Dacă un astfel de element nu există, atunci funcția returnează iteratorul end().

| pair<iterator, iterator> equal_range (**const** value_type& val) **const**;

Funcția returnează o pereche de iteratori. Primul iterator din pereche (first) este lower_bound(), iar al doilea iterator (second) este upper_bound().

Exemplu

Vom defini un set și un multiset pe care le inițializăm cu caracterele dintr-un sir :

```
| string Sir("abracadabra");
set<char> s(Sir.begin(), Sir.end());
multiset<char> ms(Sir.begin(), Sir.end());
```

Putem parcurge mulțimea s și multimulțimea ms, afișând caracterele :

```
| set<char>::iterator s_it;
multiset<char>:: iterator ms_it;
for (s_it=s.begin(); s_it!=s.end(); ++s_it)
    cout<<*s_it;
cout<<'\n';
for (ms_it=ms.begin(); ms_it!=ms.end(); ++ms_it)
    cout<<*ms_it;
cout<<'\n';
```

Pe ecran se va afișa :

```
abcdr
aaaaabbcdr
```

Putem determina `lower_bound()` și `upper_bound()` pentru caracterul b în ambele containere :

```
s_it=s.lower_bound('b'); ms_it=s.lower_bound('b');
cout<<*s_it<<*ms_it<<'\n';
s_it=s.upper_bound('b'); ms_it=s.upper_bound('b');
cout<<*s_it<<*ms_it<<'\n';
```

Pe ecran se va afișa :

```
bb
cc
```

Dacă executăm aceeași secvență pentru caracterul p (care nu apare în sir), obținem :

```
rr
rr
```

Aplicația 1. Sea

Pe mare va avea loc o importantă bătălie între N vapoare. Vapoarele sunt considerate niște puncte și sunt date prin coordonatele lor carteziene x și y. Din motive greu de înțeles, vapoarele nu pot ataca decât vapoarele care se află la stânga și mai jos (mai exact, un vapor la poziția x_1, y_1 poate ataca alt vapor la poziția x_2, y_2 dacă și numai dacă $x_1 > x_2$ și $y_1 > y_2$). Pentru că această bătălie are loc în zona Triunghiului Bermudelor, vapoarele apar (se teleportează) pe rând în zona bătăliei. Vapoarele sunt numerotate 1, 2, ..., N în ordinea apariției lor. În momentul în care un vas apare, dacă există alt vas care a apărut deja și care poate să îl atace pe cel nou, vasul nou este distrus instantaneu. Dacă nu, vasul cel nou rămâne pe mare și distrugă toate vasele pe care le poate ataca.

Cerință

Dându-se coordonatele la care apar pe rând vapoarele, să se afle pentru fiecare vapor dacă este distrus sau nu în momentul apariției sale, și dacă nu este distrus, să se precizeze numărul total de vapoare rămase pe mare după apariția sa.

Date de intrare

Pe prima linie a fișierului de intrare `sea.in` se află numărul natural N reprezentând numărul de vapoare ce vor apărea pe mare. Pe fiecare dintre următoarele N linii sunt descrise în ordine pozițiile vapoarelor. Mai exact pe cea de a i-a linie dintre cele N se află două numere întregi separate prin spațiu reprezentând coordonatele x și ale vaporului i ($1 \leq i \leq N$).

Date de ieșire

Fișierul de ieșire `sea.out` va conține N linii. Pe linia i se va scrie -1 dacă vaporul i este distrus în momentul apariției sau un număr întreg strict pozitiv reprezentând numărul de vapoare de pe mare după apariția vaporului i în caz contrar.

Restricții

- $1 \leq N \leq 200000$
- Coordonatele sunt numere întregi strict pozitive mai mici sau egale cu 260000
- Nu vor exista două vase cu aceeași coordonată x sau cu aceeași coordonată y .

Exemple

<code>sea.in</code>	<code>sea.out</code>	<i>Explicații</i>
5	1	Vaporul 1 rămâne pe mare
4 1	1	Vaporul 2 rămâne pe mare și distrugă vaporul 1
8 6	-1	Vaporul 3 este distrus de vaporul 2
7 5	-1	Vaporul 4 este distrus de vaporul 2
3 4	2	Vaporul 5 rămâne pe mare, împreună cu vaporul 2
9 3		

Pregătirea lotului național de informatică 2004

Soluție

Un vas va fi reprezentat prin perechea coordonatelor sale (x, y) . Mulțimea vaselor o vom reprezenta ca un set. Perekile vor fi ordonate implicit strict crescător după coordonata x (abscisă). Cum nu există două vase cu aceeași abscisă, la ordonare coordonata y nu contează.

```
| set< pair<int,int> > Vase;
```

Când citim un vas (x, y) , căutăm în set primul vas (x_1, y_1) pentru care $x_1 > x$ (acesta se obține în timp logaritmic cu funcția `upper_bound()`).

Dacă un astfel de vas există și ordonata sa $y_1 > y$, atunci acest vas va distrugă vasul (x, y) și afișăm -1 . Să observăm că este suficient să facem comparația cu $(x_1, y_1) = \text{upper_bound}()$. Dacă am presupune prin reducere la absurd că există un alt vas (x_2, y_2) cu $x_2 > x_1 > x$ și $y_2 > y > y_1$, atunci vasul (x_2, y_2) ar fi trebuit să distrugă vasul (x_1, y_1) , deci acesta ar fi fost eliminat din set la un pas anterior.

Să analizăm acum ce se întâmplă dacă (x, y) nu este distrus de un alt vas existent în set. În acest caz, va trebui să eliminăm din set toate vasele pe care vasul (x, y) le distrugă. Vom parcurge mulțimea în sens invers, începând de la `upper_bound()` spre început (aici sunt vasele (x_1, y_1) cu $x_1 < x$) și le vom șterge (cu funcția `erase()`) pe cele pentru care $y_1 < y$.

Și aici trebuie să mai facem o observație: când întâlnim un vas (x_1, y_1) pentru care $x_1 < x$ și $y_1 > y$, nu are rost să continuăm parcurgerea, ne oprim, deoarece înaintea acestui vas nu mai pot exista vase care ar putea fi distruse de (x, y) .

(deoarece ar fi fost distruse de (x_1, y_1) la un pas anterior). Deducem că vasele care trebuie eliminate formează un domeniu (care se termină cu `upper_bound()`) și pot fi șterse toate deodată.

Implementarea utilizând tipul `set` din *STL* devine foarte simplă și are complexitatea $O(N \log N)$.

```
#include <cstdio>
#include <set>
using namespace std;

set< pair<int,int> > Vase;
int main()
{ FILE *fin = fopen("sea.in", "r");
  FILE *fout = fopen("sea.out", "w");
  int x, y, n, i;
  set< pair<int,int> ::iterator It, It2, It1;
  fscanf(fin, "%d", &n);
  for (i=0; i<n; i++)
  {
    fscanf(fin, "%d %d", &x, &y);
    It = Vase.upper_bound(make_pair(x, y));
    if (It != Vase.end() && It->second > y)
      { fprintf(fout, "-1\n"); continue; }
    if (!Vase.empty())
      { for (It1=It2=It, --It1;
            It2!=Vase.begin() && It1->second < y;
            It2 = It1, --It1);
        Vase.erase(It2, It);
      }
    Vase.insert(make_pair(x, y));
    fprintf(fout, "%d\n", Vase.size());
  }
  fclose(fout);
  return 0;
}
```

Aplicația 2. Micro

Meșterul Vasile are o firmă de microprocesoare. Pe 1 ianuarie el își planifică activitatea pentru următoarele n zile. Analizând graficul de producție pentru cele n zile, Vasile știe că în ziua i firma sa poate produce cel mult p_i microprocesoare, costul de producție al unui microprocesor fiind cp_i ($1 \leq i \leq n$). Analizând situația comenzilor, Vasile știe că în ziua i el trebuie să livreze exact nr_i microprocesoare ($1 \leq i \leq n$). Microprocesoarele care nu sunt livrate pot fi trimise la depozit pentru a fi utilizate ulterior. În noaptea dintre zilele i și $i+1$ pot fi depozitate cel mult d_i microprocesoare, costul depozitării unui microprocesor fiind cd_i ($1 \leq i < n$).

Cerință

Scripteți un program care să determine care este suma minimă pe care meșterul Vasile trebuie să o cheltuiască pentru producție și depozitare astfel încât să satisfacă toate comenziile pentru cele n zile.

Date de intrare

Fișierul de intrare `micro.in` conține pe prima linie numărul natural n. Pe următoarele n linii se află informații despre producție și comenzi. Mai exact, pe cea de a i-a linie dintre cele n se află trei numere naturale separate prin spații: $p_i \ c_{p_i} \ nr_i$ ($1 \leq i \leq n$). Pe următoarele $n-1$ linii se află informații despre depozitare. Mai exact, pe linia i dintre cele $n-1$ ($1 \leq i \leq n-1$) se află două numere naturale separate prin spațiu $d_i \ cd_i$.

Date de ieșire

Fișierul de ieșire `micro.out` va conține o singură linie pe care va fi scris un singur număr întreg: suma minimă necesară pentru satisfacerea tuturor comenziilor sau valoarea -1 dacă acest lucru nu este posibil.

Restricții și precizări

- $1 \leq n \leq 100000$
- $0 \leq p_i, c_{p_i}, nr_i \leq 10^9$, pentru $1 \leq i \leq n$
- $0 \leq d_i, cd_i \leq 10^9$, pentru $1 \leq i \leq n-1$

Exemplu

<code>micro.in</code>	<code>micro.out</code>	<i>Explicație</i>
3 10 4 1 2 2 6 11 10 8 7 3 3 5	116	<p>În prima zi pot fi produse maxim 10 microprocesoare, la 4 lei microprocesorul, și trebuie să fie livrat un singur microprocesor. Cost pentru ziua 1: 4 lei. Peste noapte pot fi stocate doar 7 microprocesoare, la costul de 3 lei bucata.</p> <p>A doua zi sunt produse două microprocesoare cu 2 lei bucata și trebuie să fie livrate șase. Vom livra cele două microprocesoare produse plus 4 din depozit (stocate din prima zi). Cost pentru ziua 2: $2*2 + (4+3)*4 = 32$.</p> <p>A treia zi pot fi produse maxim 11 microprocesoare la costul de 10 lei bucata și trebuie să fie livrate 8 microprocesoare. Este preferabil să livrăm 8 procesoare produse în ziua 3 cu 10 lei bucata, adică în total 80 lei. Cost total: $4 + 32 + 80 = 116$.</p>

FII Competition, 2011

Soluție

Metoda utilizată este *Greedy*.

Parcurgem în ordine cele n zile. Pe parcurs construim un multiset de perechi (cost, cantitate). La trecerea de la ziua i către ziua $i+1$, trebuie adunată la toate elementele din multiset valoarea cd_i . În ziua i adăugăm la multiset costul de producție c_{p_i} și extragem din multiset varianta optimă de a satura cererea nr_i .

De asemenea, trebuie să luăm în considerare și restricțiile de depozitare: pentru a nu depăși d_1 , extragem din multiset cele mai costisitoare elemente.

Programul complet arată astfel:

```
#include <cstdio>
#include <vector>
#include <set>
#define NMAX 100000
#define VMAX 1000000000
#define PII pair<long long, int>
#define mp make_pair
using namespace std;

int main()
{freopen("micro.in", "rt", stdin);
freopen("micro.out", "wt", stdout);
int n, i, ;
scanf("%d", &n);
vector<int> p(n), cp(n), nr(n), d(n), cd(n);
for (i=0; i<n; i++)
    scanf("%d %d %d", &p[i], &cp[i], &nr[i]);
for (i=0; i<n-1; i++)
    scanf("%d %d", &d[i], &cd[i]);

multiset<PII > q;
long long costdep = 0, total = 0, rez = 0;
for (i=0; i<n; i++) //parcurg in ordine cele n zile
{q.insert(mp(cp[i]-costdep, p[i]));
//adaug microprocesoarele produse in ziua i
total += p[i];
//livrez cele nr[i] microprocesoare necesare
while (!q.empty() && nr[i] > 0)
{
    //parcurg multisetul q
    multiset<PII >::iterator it = q.begin();
    PII source = *it;
    int cete = min(nr[i], source.second);
    //voi livra cete microprocesoare
    source.second -= cete;
    nr[i] -= cete;
    rez += cete * (costdep + source.first);
    //adun costul microprocesoarelor livrate

    q.erase(it);

    if (source.second > 0) q.insert(source);
    total -= cete;
}}
```

```

if (nr[i] > 0) //problema nu are solutie
    {rez = -1; break; }

//adun costurile de depozitare
if (i != n-1)
    {costdep += cd[i];
     //extrag din depozit elemente cat timp depasesc
     //capacitatea de depozitare
     while (total > d[i])
         {multiset<PII>::iterator it = q.end();
          --it;
          PII last = *it;
          q.erase(it);
          long long cate = total-d[i];
          total -= last.second;
          if (last.second > cate)
              {
                  last.second -= int(cate);
                  q.insert(last);
                  total += last.second;
              }
         }
    }
printf("%lld\n", rez);
return 0;
}

```

Aplicația 3. Egal

Toată lumea îl știe pe TractoMarm, cel care numără noduri din arbori mai repede decât își numără alții degetele de la mâini. Zilele trecute, TractoMarm s-a gândit să se „joace” cu N seifuri. Cele N seifuri sunt dispuse sub forma unui arbore (graf conex aciclic). Considerăm vârful 1 ca rădăcină a arborelui. Fiecare seif are asociată o cheie cu care seiful poate fi deschis. Din păcate (sau din fericire), pot exista două sau mai multe seifuri care pot fi deschise cu aceeași cheie.

Cerință

Scriți un program care, pentru fiecare seif x , determină care este cheia care deschide cele mai multe seifuri din subarborele cu rădăcina în x .

Date de intrare

Fișierul de intrare `egal.in` conține pe prima linie numărul natural N , reprezentând numărul de seifuri. Pe următoarele $N-1$ linii se găsesc câte două numere x , semnificând că există muchie în arbore între seiful x și seiful y . Pe ultima linie se află N numere naturale, al i -lea număr reprezentând cheia care deschide seiful i .

Date de ieșire

În fișierul de ieșire `egal.out` veți afișa N linii. Linia i va conține două numere naturale separate prin spațiu, reprezentând cheia care deschide cele mai multe seifuri în subarborele cu rădăcina i și de câte ori apare aceasta în subarbore.

Restricții

- $1 \leq N \leq 100\ 000$
- În cazul în care există mai multe chei cu același număr maxim de apariții se va afișa cea minimă;
- Memoria disponibilă pentru stivă este de maxim 8 MB !

Exemplu

<code>egal.in</code>	<code>egal.out</code>
7	1 3
1 2	3 1
1 3	2 3
3 4	2 1
3 5	1 2
5 6	1 1
5 7	1 1
1 3 2 2 2 1 1	

Algoritmiada 2011

Soluție

Pentru fiecare nod x din arbore construim un container de tip `map` M, unde $M[c] = nr$, dacă în subarborele cu rădăcina x cheia c apare de nr ori. Pentru aceasta, vom parcurge arborele DFS începând cu vârful 1. După ce toți fiile vârfului x au fost procesați, putem construi și containerul `map` pentru vârful x , unificând containerele `map` asociate fiilor. E posibil să depășim memoria disponibilă, prin urmare, după ce unificăm containerele fiilor, le putem goli. Dar pentru aceasta, la fiecare nod x , după ce am calculat containerul `map` corespunzător, determinăm și reținem cheia cu număr maxim de apariții.

```
#include <cstdio>
#include <vector>
#include <map>
#define NMAX 100001
using namespace std;
int N;
vector<int> G[NMAX];
map<int, int> M[NMAX];
int c[NMAX], cmax[NMAX], nr[NMAX];
vector<bool> viz(NMAX, false);
map<int, int>::iterator it;

void citire();
void DFS(int);
```

```

void afisare();

int main()
{citire();
 DFS(1);
 afisare();
 return 0;
}

void citire()
{FILE * fin=fopen("egal.in","r");
 int i, x, y;
 fscanf(fin,"%d", &N);
 for (i=0; i<N-1; i++)
 { fscanf(fin,"%d %d", &x, &y);
   G[x].push_back(y); G[y].push_back(x);
 }
 for (i=1; i<=N; i++) fscanf(fin,"%d", &c[i]);
}

void DFS(int x)
{int i, maxim, chmax;
 viz[x]=true;
 for (i=0; i<G[x].size(); ++i)
 {if (!viz[G[x][i]])
    {DFS(G[x][i]);
     for (it=M[G[x][i]].begin();
          it!=M[G[x][i]].end(); ++it)
      M[x][it->first]+=it->second;
     M[G[x][i]].clear();
    }
 }
 M[x][c[x]]++;
 maxim=0;
 for (it=M[x].begin(); it!=M[x].end(); ++it)
 if (it->second>maxim)
   {maxim=it->second; chmax=it->first; }
 cmax[x]=chmax; nr[x]=maxim;
}

void afisare()
{
 FILE * fout=fopen("egal.out", "w");
 int x;
 for (x=1; x<=N; x++)
   fprintf(fout,"%d %d\n", cmax[x], nr[x]);
 fclose(fout);
}

```

7.2. Containerele asociative sortate map și multimap

Containerul `map` stochează elemente constituite dintr-o combinație formată dintr-o cheie și o valoare asociată cheii. Elementele sunt identificate în mod unic prin intermediul cheii (cu alte cuvinte cheile sunt distințe) și sunt ordonate crescător după cheie, conform unui criteriu de sortare specificat.

Containerul `multimap` este asemănător, diferența constând în faptul că cheile nu sunt neapărat distințe, pot exista mai multe elemente cu aceeași cheie.

Pentru a utiliza aceste clase trebuie să includem fișierul antet `map`:

```
#include <map>
```

Declarația clasei `map`:

```
template < class Key,
          class T,
          class Compare = less<Key>,
          class Alloc = allocator<pair<const Key, T> >
        > class map;
```

Clasa şablon `map` are patru tipuri specificate ca parametri:

- `Key` – tipul cheii;
- `T` – tipul valorilor;
- `Compare` – criteriul utilizat pentru compararea cheilor (implicit se utilizează operatorul `<`); criteriul de sortare este un predicat binar care primește ca parametri două chei (să spunem `c1, c2`) și returnează `true` dacă `c1` precedă conform criteriului de sortare pe `c2`;
- `Alloc` – tipul obiectului utilizat pentru alocarea memoriei pentru elementele containerului; implicit acesta este `allocator<pair<const Key, T> >`.

Declarația este similară și pentru `multimap`.

În cazul claselor `map/multimap` este definit tipul `value_type` reprezentând tipul elementelor din container, ca pereche cheie-valoare astfel:

```
typedef pair<const Key, T> value_type;
```

Clasele `map` și `multimap` nu permit modificarea directă a cheilor elementelor stocate în container, fără a afecta integritatea structurii. Dacă este necesară modificarea cheii, extrageți din structură elementul cu cheia veche și inserați un element cu noua cheie.

Iteratorii containerelor `map/multimap` sunt bidirecționali.

Deși standardul *STL* nu specifică explicit acest lucru, elementele containerelor `map/multimap` sunt organizate intern ca un arbore binar de căutare echilibrat. Acest lucru permite ca operațiile de căutare, inserare și extragere să se realizeze în timp logaritmic.

Constructorii și destructorul

Pentru clasele map/multimap sunt definiți constructorul implicit (creează un container vid), constructorul bazat pe un domeniu (creează un container în care copiază elementele dintr-un domeniu specificat), constructorul de copiere și destructorul, care distrug obiectul, eliberând memoria alocată dinamic. Începând cu C++ 11, sunt definiți constructorul de inițializare (creează un container în care plasează elementele specificate într-o listă de inițializare) și constructorul de mutare (creează un container în care mută elementele dintr-un container specificat).

Exemple

| `map<string, int> C;`

Creează un container de tip map în care cheile sunt de tip string, iar valorile asociate cheilor sunt de tip int. Elementele sunt stocate în ordinea lexicografică a cheilor.

| `map<int, string, CompInt> c2;`

Creează un container de tip map în care cheile sunt de tip int, iar valorile asociate cheilor sunt de tip string. Elementele sunt stocate în ordinea cheilor definită de clasa CompInt, care permite compararea a doi întregi în funcție de ultima cifră (definită în secțiunea precedentă).

Funcțiile membre referitoare la dimensiunea containerului

Pentru clasele map/multimap sunt definite funcțiile `empty()`, `size()` și `max_size()`, cu semnificația uzuală.

Accesori

Containerele map/multimap conțin funcțiile membre `begin()`, `end()`, `rbegin()`, `rend()` care returnează iteratori cu semnificația uzuală. Începând cu C++ 11, sunt definite și funcțiile `cbegin()`, `cend()`, `crbegin()`, `crend()`, care returnează `const_iterator`. Reamintim că elementele containerelor map/multimap sunt ordonate după chei (conform criteriului de sortare specificat la declarare). Prin urmare, primul element al containerului este primul conform ordinii specificate („cel cu cheia cea mai mică”), iar ultimul element este ultimul conform ordinii specificate („cel cu cheia cea mai mare”).

Funcțiile membre care modifică containerul

- *Inserare*

Containerele map/multimap au supraîncărcate funcții membre pentru inserare, cu același format ca la clasele set/multiset, pentru inserarea unui singur element,

inserarea unui element, indicând o poziție de început a căutării poziției de inserare, respectiv inserarea unui domeniu. Începând cu C++ 11, sunt disponibile și inserarea cu mutare, inserarea unor elemente dintr-o listă de inițializare, precum și funcțiile `emplace()` și `emplace_hint()` care creează și inserează un element.

- **Extragere**

În clasele `map/multimap` este supraîncărcată funcția `erase()` în mod similar cu clasele `set/multiset`: ștergerea unui element a cărui poziție este specificată printr-un iterator, ștergerea elementelor dintr-un domeniu specificat și respectiv ștergerea unui element specificat prin cheia sa (în `multimap` vor fi șterse toate elementele cu cheia respectivă). Există de asemenea funcția membră `clear()` care șterge toate elementele din container.

- **Interschimbare**

Funcția membră `swap()` interschimbă containerul curent cu containerul transmis ca parametru.

Operațiile de căutare

Aceleași funcții de căutare ca la `set/multiset` sunt definite și pentru clasele `map/multimap`: `find()`, `count()`, `upper_bound()`, `lower_bound()`, `equal_range()`. În cazul claselor `map/multimap`, căutarea se realizează, evident, după cheie.

Operatorul de indexare

Un element specific containerului `map` constă în faptul că operatorul de indexare este supraîncărcat. Acest operator permite referirea directă la o valoare dintr-un container de tip `map` prin intermediul cheii asociate.

Fie `C` un container de tip `map`. Construcția `C[k]` reprezintă o referință la valoarea asociată cheii `k` (dacă o astfel de cheie există în containerul `C`). Dacă în containerul `C` nu există cheia `k`, atunci se inserează în `C` cheia `k` și este returnată o referință la valoarea asociată cheii `k`. Datorită operației de căutare efectuate, execuția acestui operator necesită timp logaritmice.

Începând cu C++ 11, este definită și funcția membră `at()`, care funcționează în mod similar (primește ca parametru o cheie și returnează valoarea asociată cheii). Diferența constă în faptul că în cazul în care cheia nu există în container, funcția `at()` „aruncă” exceptia `out_of_range`.

Exemplul 1

Vom construi un container de tip `map` în care cheile sunt denumiri ale lunilor anului, iar valoarea asociată este un număr întreg reprezentând numărul de zile din luna respectivă:

```
| map <string,int> cl;
| map <string,int>::iterator it;
```

```
c1["ianuarie"]=31;
c1["februarie"]=28;
c1["aprilie"]=30;
c1["mai"]=31;
for (it=c1.begin(); it!=c1.end(); ++it)
    cout<<it->first<<' '<<it->second<<'\n';
```

După execuția acestei secvențe de instrucțiuni pe ecran va fi afișat :

```
aprilie 30
februarie 28
ianuarie 31
mai 31
```

Observați că lunile au fost afișate în ordinea lexicografică a denumirii. Operatorul de indexare a inserat cheile în containerul `c1`, iar prin atribuire am setat valoarea asociată cheii. Să analizăm și efectul următoarei atribuirii :

```
c1["februarie"]=29;
```

Dacă veți afișa din nou conținutul containerului `c1`, veți observa că luna februarie nu a fost din nou inserată în container, ci doar s-a modificat valoarea asociată.

Exemplul 2

Vom construi același container prin inserări succesive, cu funcția `insert()` :

```
map <string,int> c2;
map <string,int>::iterator it2;
c2.insert(make_pair("ianuarie",31));
c2.insert(make_pair("februarie",28));
c2.insert(make_pair("mai",31));
c2.insert(make_pair("aprilie",31));
c2.insert(make_pair("februarie",29));
for (it2=c2.begin(); it2!=c2.end(); ++it2)
    cout<<it2->first<<' '<<it2->second<<'\n';
```

Executând această secvență veți observa că pentru luna "februarie" se afișează 28 (deoarece cheia "februarie" există deja în container, nu a mai fost inserată o dată, întrucât în `map` cheile sunt distințte).

Dacă veți înlocui în declarații `map` cu `multimap`, după executarea secvenței de instrucțiuni, obțineți pe ecran :

```
aprilie 31
februarie 28
februarie 29
ianuarie 31
mai 31
```

Acest lucru se întâmplă deoarece într-un container de tip `multimap` putem insera de mai multe ori aceeași cheie.

Exemplul 3

Dacă în containerul `c1` din primul exemplu am insera cheia Februarie, am observa că această cheie se inserează, deoarece comparația cheilor se realizează făcând diferență între majuscule și minuscule. Pentru a evita o astfel de situație, vom defini noi un criteriu de comparare (`CompS`) care compară sirurile, făcând abstracție de diferența dintre majuscule și minuscule :

```
class CompS
{public:
    bool operator() (const string& x, const string& y)
    {char cx[x.size()+1], cy[y.size()+1];
     strcpy(cx,x.c_str()); strcpy(cy,y.c_str());
     strlwr(cx); strlwr(cy);
     return strcmp(cx,cy)<0; }
};
```

Observați că am transformat sirurile din `string` în sir `C`, am transformat majusculele în minuscule (cu funcția `strlwr()`), apoi am realizat compararea sirurilor transformate cu `strcmp()`. Vom defini acum containerul `c3` utilizând acest criteriu de comparație, apoi vom insera valori și vom observa că lunile stocate în container sunt distincte :

```
map <string,int, CompS> c3;
map <string,int, CompS>::iterator it3;
c3["ianuarie"]=31;
c3["februarie"]=28;
c3["aprilie"]=30;
c3["mai"]=31;
c3["Februarie"]=29;
for (it3=c3.begin(); it3!=c3.end(); ++it3)
    cout<<it3->first<<' '<<it3->second<<'\n';
cout<<'\n';
```

Executând această secvență de instrucțiuni, obținem pe ecran :

```
aprilie 30
februarie 29
ianuarie 31
mai 31
```

Aplicația 4. Băcan

Băcanul din colț e un tip de treabă. În fiecare seară îmi fac cumpărăturile la el și de multe ori îl găsesc stând până târziu pentru „a face casa”. Mai exact, el analizează lista vânzărilor din ziua respectivă și totalizează aceste vânzări pe produse, obținând astfel un document denumit situația vânzărilor.

În lista vânzărilor apare o linie pentru fiecare vânzare făcută sub forma :

```
nume_produs * cantitate
```

Cantitatea este exprimată întotdeauna în unitatea de măsură specifică produsului. Numele produsului este separat de cantitate prin caracterul * (asterisc) precedat și urmat de câte un singur spațiu.

De exemplu :

Cascaval * 2
 Bere Tuborg * 6
 Varza * 1
 Bere Tuborg * 2
 Cascaval * 3

În situația vânzărilor, produsele trebuie să apară în ordine alfabetică, câte un produs pe o linie. Pe linia corespunzătoare unui produs este scris numele acestuia urmat de * și apoi de cantitatea totală vândută din produsul respectiv. Caracterul * trebuie să fie precedat și urmat de câte un singur spațiu.

De exemplu, pentru lista de vânzări precedentă situația vânzărilor arată astfel :

Bere Tuborg * 8
 Cascaval * 5
 Varza * 1

Fiindcă și eu sunt un tip de treabă și nu e mare lucru să-i fac un program care să genereze situația vânzărilor, aş vrea să fac acest lucru, dar... niciodată nu am timp.

Cerință

Scripteți un program care să citească lista vânzărilor dintr-o zi și care să genereze situația vânzărilor.

Date de intrare

Fișierul de intrare bacan.in conține pe prima linie numărul natural n, reprezentând numărul de vânzări efectuate în ziua respectivă. Urmează n linii, pe fiecare linie fiind descrisă o singură vânzare, sub forma din enunț.

Date de ieșire

Fișierul de ieșire bacan.out va conține pe prima linie numărul natural p reprezentând numărul de produse distincte vândute în ziua respectivă. Pe următoarele p linii sunt descrise produsele vândute, sub forma specificată în enunț.

Restricții și precizări

- $1 \leq n \leq 100\ 000$
- $1 \leq \text{Numărul de produse vândute} \leq 1000$
- Numele unui produs conține maxim 20 de caractere (litere mari, litere mici și spații). Evident, un nume nu începe și nu se termină cu spațiu.
- La o vânzare cantitatea nu depășește 100 de unități.

Exemplu

bacan.in	bacan.out
5	3
Cascaval * 2	Bere Tuborg * 8
Bere Tuborg * 6	Cascaval * 5
Varza * 1	Varza * 1
Bere Tuborg * 2	
Cascaval * 3	

.campion 2006

Soluție

Vom citi lista vânzărilor linie cu linie într-un string. De pe fiecare linie extragem cantitatea și denumirea produsului. Pentru aceasta identificăm poziția poz a caracterului separator *, apoi extragem subșirul s1 care începe la poziția 0 și are poz-1 caractere (acesta este denumirea), respectiv subșirul care începe la poziția poz+2 și ține până la sfârșitul sirului (aceasta este cantitatea). Convertim sirul care conține cantitatea într-un număr întreg x.

Vom construi un container de tip map V în care cheile sunt denumirile produselor, iar cantitățile sunt valorile asociate.

```
| map <string,int> V;
```

La fiecare linie citită actualizăm cantitatea existentă în containerul V pentru produsul respectiv :

```
| V[s1] += x;
```

Reamintim modul de funcționare a operatorului de indexare : cheia s1 este căutată în map ; dacă ea nu există este inserată ; V[s1] reprezintă o referință la valoarea asociată acestei chei.

Complexitatea va fi $O(N \log ND)$, unde N este dimensiunea listei de vânzări, iar ND dimensiunea situației vânzărilor (numărul de produse distincte).

```
#include <iostream>
#include <map>
#include <string>
#include <cstdlib>
using namespace std;

ifstream fin("bacan.in");
ofstream fout("bacan.out");

map <string,int> V;

int main()
{int n, i, poz, x;
 string s, s1;
 fin>>n; fin.get();
```

```

for (i=0; i<n; i++)
{
    getline(fin, s);
    poz=s.find('*');
    s1=s.substr(poz+2); //extrag cantitatea
    x=atoi(s1.data()); //o convertesc in intreg
    s1=s.substr(0,poz-1); //extrag numele
    V[s1]+=x; //actualizez map
}
fout<<V.size()<<'\n';
map<string,int>::iterator it;
for (it=V.begin(); it!=V.end(); ++it)
    fout<<it->first<<" * "<<it->second<<'\n';
fout.close();
return 0;
}

```

Aplicația 5. Radio

Laura a dezvoltat recent o pasiune pentru șirurile de caractere generate *aleatoriu*. Ca să o ajute să treacă mai ușor peste sesiune, prietenii ei s-au gândit să o înveselească și i-au cumpărat un astfel de șir de lungime N, conținând doar litere mici ale alfabetului englez.

De dimineață, Laura a început să asculte muzică la radio și a auzit M cuvinte, toate având aceeași lungime L, care i-au plăcut foarte mult. Aceste cuvinte sunt formate tot din litere mici ale alfabetului englez. Acum ea și-ar dori să vadă, pentru fiecare cuvânt, dacă acesta apare ca subsecvență în șirul primit cadou. Cum cuvintele sunt destul de lungi, Laura nu este sigură că le-a auzit corect, dar este convinsă că nu a înțeles greșit mai mult de K litere din fiecare cuvânt.

Așadar, voi trebui să îi spuneți, pentru fiecare din cele M cuvinte, dacă există o subsecvență de lungime L în șirul primit cadou astfel încât cuvântul și subsecvența să difere în cel mult K poziții.

Date de intrare

Pe prima linie a fișierului de intrare *radio.in* se află patru numere întregi N M L K, având semnificația din enunț. Pe următoarea linie, se află N caractere neseparate prin spații ce reprezintă șirul primit cadou de fată. Pe următoarele M linii, se află câte L caractere neseparate prin spații, ce reprezintă cuvintele pe care Laura le-a auzit la radio (așa cum le-a înțeles ea).

Date de ieșire

Fișierul de ieșire *radio.out* va conține M linii. Pe linia i ($1 \leq i \leq M$) veți afișa 1 dacă există o subsecvență în șirul primit cadou de fată care să difere în cel mult de K poziții de al i-lea cuvânt din fișierul de intrare, respectiv 0, în caz contrar.

Restricții și precizări

- $1 \leq N \leq 1\ 000\ 000$
- $1 \leq M \leq 500$
- $1 \leq K \leq 50$
- $500 \leq L \leq 2500$
- Prinț-un sir de litere mici generat aleator se înțelege un sir în care pe fiecare poziție, oricare dintre litere are aceeași probabilitate de apariție.

Exemplu

radio.in	radio.out
10 3 6 2	0
anaaremere	1
roaane	1
aareme	
renere	

Pregătirea lotului național de informatică, 2010

Soluție

Vom analiza succesiv cele M cuvinte. Să notăm cu S sirul de lungime N și cu C cuvântul curent (de lungime L). Vom denumi distanță dintre două secvențe de aceeași lungime numărul de poziții pentru care caracterele din cele două secvențe sunt diferite.

O primă idee ar fi să verificăm pentru fiecare subsecvență de lungime L a sirului S dacă distanța dintre C și subsecvența respectivă este $\leq K$. Având în vedere lungimea sirului S și numărul de cuvinte ce trebuie testate, această abordare este mult prea costisitoare ca timp.

Încercăm prin urmare să reducem numărul de comparații între secvențe. Pentru aceasta, vom împărți cuvântul curent C în $K+1$ subsecvențe de lungime egală cu $L_g = L / (K+1)$ (eventualele litere rămasă la sfârșit le vom ignora). Pentru a reduce numărul de comparații între secvențe, vom compara valorile *hash* ale secvențelor și, doar în caz de egalitate a valorilor *hash*, vom compara secvențele, pentru a calcula distanța dintre acestea.

Prin urmare vom calcula valoarea *hash* pentru fiecare secvență de lungime L_g din sirul S . Vom construi un container de tip *map* în care pentru fiecare valoare *hash* reținem pozițiile de început ale secvențelor din S care au valoarea *hash* respectivă.

```
| map< int, vector<int> > hash;
```

Pentru fiecare subsecvență de lungime L_g din cuvântul curent C calculăm valoarea *hash* și verificăm dacă această valoare *hash* apare în containerul de tip *map* *hash*. Dacă C „se potrivește” cu o subsecvență din S , obligatoriu cel puțin una dintre cele $K+1$ subsecvențe ale sale trebuie să se „potrivească” cu una dintre subsecvențele de lungime L_g ale lui S . Dacă valoarea *hash* a unei secvențe de lungime L_g din C a fost găsită în containerul *hash*, atunci parcurgem vectorul asociat acestei valori *hash* și verificăm pentru fiecare poziție de început din listă dacă secvența de lungime L care începe la poziția respectivă se „potrivește” cu cuvântul curent (în sensul că distanța este $\leq K$).

Faptul că sirul inițial este generat aleator garantează că pentru fiecare valoare *hash* nu vor exista multe poziții de început corespunzătoare, deci nu se vor face multe comparații între secvențe.

```
#include <cstdio>
#include <map>
#include <vector>
#define NMAX 1000005
#define MOD 666013
#define LMAX 2505
#define BASE 37
using namespace std;

int N, M, L, K;
char S[NMAX];
map< int, vector<int> > hash;
char C[LMAX];
int dist(char *a, char *b, int n);
void buildHash(int Lg);
bool SePotriveste(int poz);

int main()
{ freopen("radio.in", "r", stdin);
  freopen("radio.out", "w", stdout);
  vector<int>::iterator it;
  int ic, i, j, hashVal, rez;
  scanf("%d %d %d %d", &N, &M, &L, &K);
  scanf("%s", S);
  int Lg = L/(K+1);
  buildHash(Lg);
  for (ic = 0; ic < M; ++ic)
    { scanf("%s", C);
      for (rez=i=0; i+Lg<=L; i+=Lg)
        { for (hashVal=j=0; j<Lg; ++j)
            hashVal=(hashVal*BASE+(C[i+j]-'a'))%MOD;
          if (hash.find(hashVal) != hash.end())
            for (it=hash[hashVal].begin();
                  it!=hash[hashVal].end(); ++it)
              rez |= SePotriveste(*it - i);
        }
      printf("%d\n", rez);
    }
}

int dist(char *a, char *b, int n)
{int sol=0;
 for (int i=0; i<n; ++i)  sol+=a[i]!=b[i];
 return sol;
}
```

```

void buildHash(int Lg)
{int hashVal = 0, i, coef=1;
for (i=0; i<Lg-1 && i<N; ++i)
    hashVal=(hashVal * BASE + (S[i] - 'a')) % MOD;
for (i=1; i<Lg; ++i)  coef = coef * BASE % MOD;
for (i=Lg-1; i<N; ++i)
    { hashVal = (hashVal*BASE + (S[i]-'a')) % MOD;
      hash[hashVal].push_back(i-Lg+1);
      hashVal = (hashVal - coef*(S[i-Lg+1]-'a')) % MOD;
      if (hashVal < 0) hashVal += MOD;
    }
}
bool SePotriveste(int poz)
{  if (poz < 0 || poz + L > N) return 0;
  return dist(S + poz, C, L) <= K;
}

```

Putem implementa aceeași soluție utilizând un multimap în care inserăm fiecare pereche formată dintr-o valoare *hash* și poziția de început corespunzătoare.

```
| multimap< int, int > hash;
```

În acest mod nu mai formăm un vector de poziții de început pentru fiecare valoare *hash*. În funcția *buildHash()* singura modificare fiind modul de inserare în container :

```

void buildHash(int Lg)
{int hashVal = 0, i, coef=1;
for (i = 0; i < Lg-1 && i < N; ++i)
    hashVal = (hashVal * BASE + (S[i] - 'a')) % MOD;
for (i = 1; i < Lg; ++i)
    coef = coef * BASE % MOD;
for (i = Lg-1; i < N; ++i)
    { hashVal = (hashVal*BASE + (S[i]-'a')) % MOD;
      hash.insert(make_pair(hashVal,i-Lg+1));
      hashVal = (hashVal - coef*(S[i-Lg+1]-'a')) % MOD;
      if (hashVal < 0) hashVal += MOD;
    }
}

```

Când căutăm o valoare *hash* în multimap, vom utiliza funcțiile *lower_bound()* și *upper_bound()* pentru a determina domeniul de apariție al valorii (în mod echivalent se putea utiliza funcția *equal_range()*). Apoi parcurgem acest domeniu și verificăm secvențele pentru fiecare poziție de început din domeniu. Funcția *main()* se modifică astfel :

```

int main()
{ freopen("radio.in", "r", stdin);
  freopen("radio.out", "w", stdout);
  multimap< int, int >::iterator it, prim, ultim;
  int ic, i, j, hashVal, rez;
  scanf("%d %d %d %d", &N, &M, &L, &K);

```

```

scanf("%s", S);
int Lg = L/(K+1);
buildHash(Lg);
for (ic = 0; ic < M; ++ic)
{ scanf("%s", C);
  for (rez=i=0; i+Lg<=L; i+=Lg)
    { for (hashVal=j=0; j<Lg; ++j)
        hashVal=(hashVal*BASE+(C[i+j]-'a'))%MOD;
      prim=hash.lower_bound(hashVal);
      ultim=hash.upper_bound(hashVal);
      for (it=prim; it != ultim; ++it)
        rez |= SePotriveste(it->second - i);
    }
  printf("%d\n", rez);
}
}

```

7.3. Containerele asociative nesortate

Pentru a utiliza containerele asociative nesortate `unordered_set` și `unordered_multiset`, trebuie să includem fișierul antet `unordered_set`:

```
#include <unordered_set>
```

În mod similar, pentru `unordered_map` și `unordered_multimap`, trebuie să includem fișierul antet `unordered_map`:

```
#include <unordered_map>
```

În general, toate considerațiile referitoare la containerele asociative sortate rămân valabile, diferențele provenind de la organizare internă a containerelor asociative nesortate. Vom menționa în continuare doar aspectele prin care diferă:

1. Elementele dintr-un container asociativ nesortat, evident, nu sunt menținute într-o anumită ordine.
2. Intern, elementele unui container asociativ nesortat sunt organizate sub forma unei tabele de dispersie (tabelă *hash*). Astfel elementele sunt grupate în blocuri denumite *buckets*, în funcție de valoarea *hash* calculată pentru cheia elementului respectiv (reamintim că pentru containerele `set/multiset` cheia și valoarea coincid).
3. Datorită modului de organizare sub forma tabelelor de dispersie, accesul la un element individual al containerului este foarte rapid (operațiile fiind mai eficiente decât în cazul containerelor asociative sortate), însă parcurgerea unui domeniu se realizează mai puțin eficient.
4. Iteratorii claselor asociative nesortate sunt iteratori de înaintare.

Declararea containerelor asociative nesortate

Declararea containerelor asociative nesortate diferă de declararea containerelor asociative sortate datorită modului diferit de organizare internă (nu mai este specificat criteriul de sortare, în schimb trebuie specificată funcția hash și un predicator care să permită identificarea egalității cheilor).

Deoarece declarațiile containerelor asociative nesortate sunt asemănătoare, vom exemplifica doar declarațiile pentru `unordered_set` și `unordered_map`:

```
template < class Key,
           class Hash = hash<Key>,
           class Pred = equal_to<Key>,
           class Alloc = allocator<Key> >
class unordered_set;
```

Parametrii care intervin în declarație sunt:

- `Key` – tipul cheilor/valorilor stocate în container;
- `Hash` – functor unar (sau pointer la o funcție) care primește ca parametru o valoare de același tip cu cheia și returnează valoarea `hash` calculată pentru cheia respectivă; implicit este utilizat functorul standard `hash()` (declarat în fișierul `antet functional`);
- `Pred` – predicator binar care primește ca parametri două valori de același tip cu cheile și returnează `true` dacă cele două chei primite ca parametri sunt considerate echivalente; implicit este utilizat functorul standard `equal_to()` (declarat în fișierul `antet functional`), care utilizează comparația cu `==`;
- `Alloc` – definește modelul de alocare a memoriei; implicit este utilizată clasa `allocator`.

Declarația clasei `unordered_map`:

```
template < class Key,
           class T,
           class Hash = hash<Key>,
           class Pred = equal_to<Key>,
           class Alloc = allocator< pair<const Key, T> > >
class unordered_map;
```

Observați că în declarația clasei `unordered_map` intervine încă un tip (`T`) reprezentând tipul valorilor asociate cheilor.

Constructorii și destructorul

Pentru containerele asociative nesortate sunt supraîncărcați constructorul implicit, constructorul de copiere, de mutare, constructorul bazat pe un domeniu și constructorul bazat pe o listă de inițializare.

Exemple

```
| unordered_set<string> c1;
```

Creează un obiect de tip `unordered_set` denumit `c1` cu 0 elemente de tip `string`; este utilizat functorul `hash()` implicit și functorul `equal_to` pentru compararea cheilor.

```
| unordered_set<string> c2 ( {"ana", "ion", "dan"} );
```

Creează un obiect de tip `unordered_set` denumit `c2` inițializat cu 3 elemente de tip `string`; este utilizat functorul `hash()` implicit și functorul `equal_to` pentru compararea cheilor.

```
| unordered_set<string> c3 (c2.begin(), c2.end());
```

Creează un obiect de tip `unordered_set` denumit `c3` inițializat cu elementele din domeniul specificat.

```
| unordered_set<string> c4 (c2);
```

Creează obiectul de tip `unordered_set` denumit `c4`, copiind elementele din `c2`.

Funcțiile membre publice

Containerele asociative nesortate au o interfață similară cu containerele asociative sortate. Există aceleași funcții pentru determinarea dimensiunii și aceleași funcții de modificare a containerului. Datorită faptului că iteratorii sunt iteratori de înaintare, sunt disponibile numai funcțiile `begin()`, `end()`, `cbegin()`, `cend()`. În ceea ce privește funcțiile de căutare, sunt disponibile funcțiile `find()`, `count()` și `equal_range()`.

Există încă plus funcții referitoare la tabela *hash*:

```
| size_type bucket_count() const noexcept;
```

Returnează numărul de blocuri (*buckets*) existente în container.

```
| size_type max_bucket_count() const noexcept;
```

Returnează numărul maxim de blocuri (*buckets*) ce pot exista în container.

```
| size_type bucket_size ( size_type n ) const;
```

Returnează numărul de elemente din blocul cu numărul `n` din container (`n < bucket_count()`).

```
| size_type bucket ( const key_type& k ) const;
```

Returnează numărul blocului din care face parte cheia `k`.

```
| float load_factor() const noexcept;
```

Returnează factorul de încărcare a containerului (raportul dintre numărul de elemente din container și numărul de blocuri existente). Factorul de încărcare determină probabilitatea de apariție a coliziunilor în tabela de dispersie.

```
| float max_load_factor() const noexcept;
| void max_load_factor ( float z );
```

Prima formă returnează factorul maxim de încărcare a tabelei de dispersie. A doua formă stabilește z ca factorul maxim de încărcare a tabelei de dispersie. În cazul în care factorul de încărcare curent este $>z$, se reconstruiește tabela de dispersie (*rehash*). Implicit factorul maxim de încărcare este 1.0.

```
| void rehash( size_type n );
```

Dacă numărul de blocuri (*buckets*) este $\geq n$, se reconstruiește tabela de dispersie (*rehash*) astfel încât numărul de *buckets* să fie $< n$. Reconstrucția tabelei de dispersie se realizează automat ori de câte ori factorul de încărcare depășește factorul maxim de încărcare. Complexitatea acestei operații este liniară în medie, dar poate fi pătratică în cazul cel mai defavorabil.

```
| void reserve ( size_type n );
```

Modifică numărul de blocuri din container astfel încât să fie adecvat pentru a reține cel puțin n elemente. Este posibil să apară necesitatea reconstruirii tabelei de dispersie (dacă $n > \text{bucket_count} * \text{max_load_factor}$).

7.4. Exerciții și probleme propuse

1. Să considerăm următoarea secvență de operații :

```
C.insert(10);C.insert(1);C.insert(5);
C.insert(1);C.insert(-30);
for (it=C.begin(); it!=C.end(); ++it) cout<<*it;
```

Ce se va afișa pe ecran în cazul în care declarația obiectului *C* este :

- a. `set<int> C;`
- b. `multiset<int> C;`

2. Care dintre următoarele declarații pentru obiectul *M* sunt corecte ?

a.

```
| set<int, list > M;
```

b.

```
| map<char> M;
```

c.

```
| list<string> L(5, "mama");
multimap<string, list<string> > M(L);
```

d.

```
| map<int, vector<int>, greater<int> > M;
```

3. Considerând următoarea declarație :

```
| map<int,int> C;
```

Care dintre următoarele instrucțiuni sunt corecte sintactic ?

- a. C[-50]=6;
- b. C.insert(-50,6);
- c. C.insert(make_pair(-50,6));
- d. C.push_back(-50,6);

Dar în cazul în care obiectul C este declarat astfel :

```
| multimap<int,int> C;
```

4. Care dintre următoarele afirmații sunt adevărate ?

- a. În containerele set și multiset cheile și valorile asociate coincid.
- b. Nu este permisă modificarea unei valori într-un container de tip set/multiset.
- c. Nu este permisă modificarea unei chei într-un container de tip map/multimap.
- d. Pentru toate containerele asociative sortate, elementele sunt organizate intern ca un arbore binar de căutare echilibrat.
- e. Primul element dintr-un set/multiset este cel cu valoarea cea mai mică, conform criteriului de sortare specificat la declarare.
- f. Primul element dintr-un map/multimap este cel cu cheia cea mai mică, conform criteriului de sortare specificat la declarare.
- g. Elementele unui container asociativ sortat pot fi parcuse în ambele sensuri.
- h. Operațiile de căutare, inserare și stergere pentru containere asociative sortate se execută în timp logaritmice.
- i. Accesarea unui element într-un container asociativ nesortat este mai rapidă decât accesarea unui element dintr-un container asociativ sortat.
- j. Niciuna dintre afirmațiile precedente.

5. Ce se va afișa pe ecran după execuția următoarei secvențe de instrucțiuni ?

```
map<int, deque<int>, greater<int> > M;
map <int, deque<int>, greater<int> >::iterator it;
deque<int>::iterator itd;
M[20].push_back(3);M[2].push_back(5);M[20].push_back(300);
M[200].push_back(13);M[20].push_back(1);M[2].push_back(0);
M[20].push_back(33);M[50].push_back(1);M[200].push_back(8);
for (it=M.begin(); it!=M.end(); ++it)
    {cout<<it->first<< ' ';
     for(itd=it->second.begin(); itd!=it->second.end(); ++itd)
         cout<<*itd<< ' ';
     cout<<'\n';
    }
```

6. Ce se va afișa pe ecran după execuția următoarei secvențe de instrucțiuni ?

```
multimap<string,int> M;
multimap<string,int>::iterator it;
M.insert(make_pair("Ana",1));M.insert(make_pair("Xena",2));
M.insert(make_pair("Ana",3));M.insert(make_pair("Ion",4));
M.insert(make_pair("Ana",5));M.insert(make_pair("Dan",6));
```

```

M.insert(make_pair("Dan", 7)); M.insert(make_pair("Dan", 8));
for (it=M.begin(); it!=M.end(); ++it)
    cout<<it->first<< ' '<<it->second<<'\n';
it=M.lower_bound("Ana");
cout<<it->first<< ' '<<it->second<<'\n';
it=M.upper_bound("Ana");
cout<<it->first<< ' '<<it->second<<'\n';
it=M.lower_bound("Dana");
cout<<it->first<< ' '<<it->second<<'\n';
it=M.upper_bound("Dana");
cout<<it->first<< ' '<<it->second<<'\n';

```

7. Observația că în cazul problemei *Egal* (aplicația 3) nu este necesar ca în containerul `map` elementele să fie ordonate după cheie. Implementați mai eficient soluția acestei probleme, utilizând `unordered_map`.

8. Stones

Se dau n grămezi de pietre, pe care dorim să le combinăm într-o singură grămadă. Pentru aceasta, putem alege două grămezi și să le combinăm obținând astfel o singură grămadă. Procedeul se repetă până când, în final, obținem o singură grămadă. Atunci când combinăm două grămezi, consumăm energie egală cu numărul de pietre aflate în cele două grămezi care se combină. De exemplu, dacă vom combina o grămadă formată din 3 pietre cu o grămadă formată din 5 pietre consumul de energie va fi egal cu $3+5=8$.

Cerință

Date fiind n grămezi de pietre, determinați consumul minim de energie necesară pentru a combina grămezile în una singură.

Date de intrare

Fișierul de intrare `stones.in` conține pe prima linie un număr natural T , reprezentând numărul de seturi de date de test. În continuare, pe liniile $2i$ și $2i+1$ este descris al i -lea set de date de test. Pe linia $2i$ este scris numărul natural n , iar pe linia $2i+1$ sunt n numere naturale separate prin spații, reprezentând dimensiunile celor n grămezi.

Date de ieșire

În fișierul de ieșire `stones.out` se vor scrie T linii, câte una pentru fiecare set de date de test. Pe a i -a linie se va scrie un număr natural reprezentând consumul minim de energie necesară pentru combinarea grămezilor din al i -lea set de date.

Restricții

- $0 < n \leq 100000$
- Numărul de pietre din orice grămadă este ≤ 100 .

Exemplu

stones.in	stones.out
1	45
4	
5 9 6 3	

Universitatea TIANJIN, Concurs de selecție ACM 2010

9. Pikachu

Miruna și partenerul ei de aventură, Pikachu, sunt în fața unei noi provocări. Cele două personaje au ajuns lângă un lanț muntos format din N vârfuri așezate în linie dreaptă unul după altul. Pentru fiecare vârf muntos se cunoaște înălțimea lui. Folosindu-se de puterile sale extraordinare, Pikachu este capabil să scadă sau să crească înălțimea unui vârf muntos cu o unitate într-o secundă. Din motive necunoscute muritorilor de rând, cei doi prieteni vor să obțină cel puțin K vârfuri montane consecutive care au aceeași înălțime într-un timp cât mai scurt.

Cerință

Determinați timpul minim în care Pikachu poate îndeplini această sarcină.

Date de intrare

Fișierul de intrare *pikachu.in* va conține pe prima linie două numere N și K având semnificația din enunț. Pe cea de-a doua linie se vor găsi N numere naturale reprezentând înălțimile vârfurilor muntoase.

Date de ieșire

Fișierul de ieșire *pikachu.out* va conține un singur număr natural T, reprezentând timpul minim necesar pentru a obține cel puțin K vârfuri consecutive cu aceeași înălțime.

Restricții și precizări

- $1 \leq N \leq 105$
- $1 \leq K \leq N$
- Înălțimile munților sunt numere pozitive care se pot reprezenta pe întregi de 32 de biți cu semn
- Rezultatul se poate reprezenta pe un întreg de 64 de biți cu semn

Exemplu

pikachu.in	pikachu.out	Explicație
5 3 5 2 4 3 7	2	În prima secundă se crește înălțimea muntelui de pe poziția a doua, iar în a doua secundă se scade înălțimea muntelui de pe poziția a treia. În urma acestor operații subsecvența care începe pe a doua poziție și se termină pe a patra poziție va conține doar vârfuri de înălțime 3.

10. Submatrix

Miruna a găsit pe fundul mării o matrice cu n linii și m coloane având elementele numere naturale. Din motive necunoscute, Mirunel vrea să afle care este cea mai mare submatrice pătratică ce conține maxim k numere distincte.

Cerință

Scrieți un program care să determine latura maximă a unei submatrice care respectă condițiile lui Mirunel.

Date de intrare

Fișierul de intrare `submatrix.in` conține pe prima linie trei numere naturale n m k separate prin câte un singur spațiu având semnificația din enunț. Pe următoarele n linii se găsesc câte m numere naturale separate prin spațiu reprezentând valorile din matrice.

Date de ieșire

Fișierul de ieșire `submatrix.out` va conține o singură linie pe care va fi scris, un singur număr natural, latura submatricei căutate.

Restricții și precizări

- $1 \leq n, m \leq 300$
- $1 \leq k \leq n * m$

Exemple

<code>submatrix.in</code>	<code>submatrix.out</code>	<i>Explicație</i>
<pre>5 7 3 6 5 7 3 6 6 7 5 7 5 5 7 3 7 3 3 5 3 5 6 7 7 7 5 5 5 6 7 7 7 6 5 6 3 5</pre>	3	Submatricea având colțul din stânga sus pe poziția $(2, 3)$ și latura 3 conține doar elementele 3, 5 și 7.

8. Soluții și indicații

1. Principiile programării orientate pe obiect

1. e; 2. c.

2. Programarea orientată pe obiecte în C++

1. c; 2. d; 3. a; 4. b, c; 5. c; 6. b; 7. a, b, c; 8. c; 9. a cu d și c cu d; 10. b;
11. d.; 12. b; 13. a, c; 14. b; 15. b; 16. b; 17. b; 18. d; 19. c; 20. c, e; 21. d;
22. d; 23. b, c; 24. c, f; 25. a.

28. Euclid

Vom începe căutarea soluției de la sfârșit: ultima împărțire trebuie să aibă restul 0. Aceste două numere vor forma primele elemente ale șirului de soluție: (e_1 și e_2). În concluzie, e_2 este divizibil cu e_1 .

Luând în considerare că ne interesează cea mai mică pereche cu această proprietate $\Rightarrow e_1=1$ și $e_2=2$.

Conform penultimului pas al algoritmului lui Euclid $e_3 \text{ mod } e_2 = e_1$. Adică ne interesează cea mai mică valoare pentru $e_3 \text{ mod } 2 = 1 \Rightarrow e_3 = 3$.

Din asta va rezulta că $e_4 \text{ mod } e_3 = e_2$. Adică $e_4 \text{ mod } 3 = 2$ și cea mai mică valoare care îndeplinește condiția este $e_4 = 5$.

Se observă că șirul se poate defini cu recurența $e_1=1$, $e_2=2$, $e_k = e_{k-1} + e_{k-2}$ și elementele sale sunt numere Fibonacci: 1, 2, 3, 5, 8, 13, ...

Această metodă garantează, pe de o parte, numărul de pași maxim în algoritmul lui Euclid, iar pe de altă parte, soluția minimă în cazul existenței mai multor perechi cu același număr de pași. Să observăm acea proprietate a algoritmului lui Euclid, conform căreia, dacă împărțim un număr mai mic cu un număr mai mare, cele două numere se vor inversa. Astfel putem câștiga încă un pas în favoarea soluției.

În concluzie, soluția finală va fi formată din ultimele două elemente ale șirului crescător $e_1 < e_2 < e_3 < \dots < e_{p-1} < e_p \leq n$. Numărul de pași va fi p .

Având în vedere mărimea datelor de intrare, problema se va rezolva pe numere mari.

3. Elemente de programare generică

1. c ; 2. d ; 3. c ;

6. Unific

a. Se construiește vectorul de frecvență al cifrelor.

b. Ideea de a parurge vectorul și de a realiza eliminările conform enunțului este ineficientă. Vom citi succesiv numerele și le vom introduce într-o stivă. La fiecare pas analizăm elementul curent și elementul din vârful stivei. Dacă este posibil, realizăm unificarea (extragem din stivă elementul de la vârf în acest caz). Nu introducem însă numărul citit (sau rezultatul obținut la precedenta unificare) decât atunci când suntem siguri că nu mai este posibilă o altă unificare (deci procedeul descris se repetă cât timp valoarea din vârful stivei și valoarea curentă se pot unifica). O atenție specială la unificare trebuie acordată cifrelor egale cu 0.

4. STL. Concepțe generale

1. c ; 2. d ; 3. b, d, g, i ;

4. Construim o funcție de generare a numerelor aleatorii (era posibilă și definirea unui functor, dar varianta cu funcția e mai simplă) :

```
| int GenT () { return (rand()%10000); }
```

Generarea unui vector v format din numere aleatorii se poate realiza cu algoritmul generate() astfel:

```
| generate(f, f+n, GenT);
```

5. Construim funcția elimin(), care elimină cifra/cifrele din mijlocul numărului :

```
void elimin(int& x)
{
    int lg, i;
    vector<int> c;
    //extrag cifrele lui x și le plasez în c în ordine inversă
    while (x) {c.push_back(x%10); x/=10;}
    //calculează lungimea lui x
    lg=c.size();
    //construiesc a doua jumătate
    for (x=0, i=lg-1; i>lg/2; i--) x=x*10+c[i];
    //sar una sau două cifre din mijloc
    i-=lg%2?1:2;
    //concatenez prima jumătate
    while (i>=0) x=x*10+c[i--];
}
```

Pentru a transforma toate valorile dintr-un vector v :

```
| for_each(v.begin(), v.end(), elimin);
```

5. Containere secvențiale

1. c ; 2. a, c, d ;

3.

3 3

7 7

6 7

amMama ;

4. b ; 5. d ;

6. Lista v1 va conține 1 1 2 3 3 4 5 7, iar lista v2 va fi vidă.

7. Lista v1 va conține 1 6 5 4 3 2 1 2 3 4 iar lista v2 va conține 7.

8. 4

9. Mama si Ada vorbe nes

10. a, c ; 11. a, c ;

15. Arme

Soluția 1. Vom utiliza metoda *Greedy*. Sortăm armele de la brâu crescător după punctaj. Sortăm și armele din camera armelor crescător după punctaj. Parcurgem în ordine crescătoare armele de la brâu și le înlocuim pe cele mai mici arme de la brâu cu cele mai mari din camera armelor (doar dacă punctajul se mărește).

Soluția 2. Utilizând interclasarea a doi vectori sortați (puterile armelor), însumăm primele n valori din cei doi vectori sortați descrescător.

Soluția 3. Se ordonează crescător sirul puterilor armelor de la brâu și descrescător sirul puterilor armelor de pe perete. Se înlocuiește arma cu cea mai mică putere cu arma cu putere maximă dintre cele aflate pe perete (dacă este posibil) și aşa mai departe până când nu mai putem alege nimic de pe perete ($pb_i \geq pc_i$) sau nu mai avem arme la dispoziție ($m < n$). Suma inițială a puterilor pb_i va crește la fiecare înlocuire cu diferența $pc_i - pb_i$.

16. Segm

Metoda utilizată este *Greedy*. Vom defini un tip denumit segment, ca fiind o clasă în care reținem două date membre: extremitatea inițială și cea finală. Vom sorta segmentele crescător după extremitatea finală; dacă există mai multe segmente cu aceeași extremitate finală, le sortăm descrescător după extremitatea inițială (pentru aceasta putem supraîncărca operatorul $<$). Vom utiliza două variabile *Last* și *PrevLast* în care reținem ultimul punct selectat în mulțime (cu abscisa maximă până la momentul curent) și respectiv penultimul punct selectat în mulțime. Parcurgem acum segmentele în ordine. Dacă *Last* (ultimul punct selectat) nu aparține segmentului curent, atunci includem în mulțime două noi puncte (extremitatea finală a segmentului curent și extremitatea finală -1). Dacă *Last* aparține segmentului curent, verificăm dacă și *PrevLast* aparține segmentului curent. Dacă nu, vom include în mulțime extremitatea finală a segmentului curent.

17. Compus

Problema se rezolvă utilizând un dequeue. Scopul structurii este de a procesa la fiecare moment elementul minim din dreapta elementului curent, pe un interval cu lungime egală cu cea a intervalului dintre începutul sirului și elementul curent. Pentru a realiza acest lucru, elementele sirului sunt parcurse de la dreapta spre stânga și introduse în dequeue prin partea dreaptă împreună cu un „timp de intrare” asociat (0 pentru elementul cel mai din dreapta, 1 pentru următorul etc.).

Elementul introdus la un moment dat „elimină” elementele din dreapta dequeue-ului care au valori mai mari sau egale cu el deoarece niciunul nu mai poate reprezenta element minim la dreapta. Pentru accesarea la un moment dat a unui minim la dreapta valid (minimul la dreapta unui element cu numărul de ordine i ($i \geq 0$) care se află suficient de aproape de i pentru a fi inclus în prefixul centrat în i), se vor elimina mai întâi din capătul din stânga elementele care au „expirat” (adică acele elemente care sunt minime la dreapta, dar se află prea departe de elementul curent pentru a intra în compoziția prefixului). Acest lucru îl realizăm utilizând timpii de intrare ai elementelor în dequeue. Timpul de intrare care ne interesează este cel al elementului imediat următor sfârșitului prefixului curent. Dacă prefixul curent este dat de mijlocul i , atunci elementul căutat este pe poziția $2*i$ în sir. Conform convenției de notare a timpilor de intrare, suma dintre timpul de intrare în dequeue și numărul de ordine din sirul inițial al unui element este $n-1$. Astfel, deducem că timpul limită de intrare căutat este $(n-1)-2*i$. În acest moment, în capătul din stânga al dequeue-ului se află minimul la dreapta valid căutat. Dacă elementul curent este mai mare decât minimul său din dreapta și este la rândul său maxim la stânga (mai mare decât toate elementele din stânga sa) atunci el determină un prefix compus.

18. Secvop

Vom utiliza o listă dublu înălțuită în care vom reține intervalele ce se obțin în urma operațiilor de inversare (un container de tip list). Inițial lista este constituită dintr-un singur element care reprezintă intervalul $[1, N]$. La fiecare operație de inversare $i \ x \ y$:

- se determină elementul situat pe poziția x ;
- se partionează intervalul în care se află elementul situat pe poziția x în două intervale;
- se determină elementul situat pe poziția y ;
- se partionează intervalul în care se află elementul situat pe poziția y în două intervale;
- se parcurg elementele listei situate între pozițiile x și y și se inversează intervalele corespunzătoare.

Pentru a calcula suma elementelor $s \ x \ y$ este suficient să parcurg toate elementele din lista corespunzătoare intervalelor situate între intervalul în care se află poziția x și intervalul în care se află poziția y , se determină suma elementelor din fiecare interval și se însumează sumele. Evident, o atenție specială trebuie să acordăm exact intervalelor în care se află poziția x și respectiv poziția y (aici nu trebuie calculată suma din tot intervalul și doar până la poziția x , respectiv y).

19. Chei

Putem asocia problemei un graf orientat astfel:

1. vârfurile grafului sunt cei N purceluși (numerotați de la 1 la N);
2. există arc de la vârful i la vârful j dacă cheia purcelușului j se află în purcelușul i (i va fi numit predecesorul lui j).

Acesta este un graf particular (fiecare vârf j are exact un predecesor – notat $pre[j]$). Vom reprezenta grafului prin liste de adiacență. De asemenea vom utiliza un vector $chei$: $chei[i]=0$, dacă purcelușul i nu a fost deschis, sau $n+1$, dacă purcelușul i a fost deschis.

Cât timp mai există purceluși care nu au fost deschiși:

- alegem primul purceluș închis;
- pornind din acest purceluș, construim „drumul” înapoi până obținem un circuit.

Pentru a „rezolva” acest circuit vom sparge un purceluș de pe circuit.

Apoi facem o parcurgere *DFS* din purcelușul spart, pentru a deschide toți purcelușii care se pot deschide cu cheile care se obțin. Complexitate (memorie și timp) : $O(N)$.

6. Clasele adaptor

1.

- a. 25 13 2 10 7
- b. 25 13 10 7 2

2.

```
S.push('a'); S.push('m');
cout<<S.top(); S.pop();
cout<<S.top();
S.push('m'); cout<<S.top(); S.pop();
cout<<S.top();
```

3. d; 4. c; 5. b;

9. Swap

a) Rezolvarea presupune folosirea unei stive în care sunt păstrate pozițiile parantezelor deschise, neîmperecheate la momentul curent. Pentru rezolvarea primului punct când este întâlnită o paranteză închisă se adaugă la costul total costul parantezei pereche formate din vârful stivei și paranteza închisă curentă. Notăm cu C_{tot} costul determinat la punctul a.

b) Costul minim care se poate obține în urma efectuării unei singure operații *swap* asupra parantezării initiale este egal cu $C_{tot} - 2$. Dacă există cel puțin o posibilitate de a efectua o operație *swap*, se va afișa $C_{tot} - 2$, altfel se va afișa -1

c) Există patru tipuri de perechi de paranteze care pot fi interschimbate :

-)) – se obține același sir deci același cost ;
- ((– se obține același sir deci același cost ;

) (– se obține un sir cu cost suplimentar 2. Orice astfel de caz este redus la situația următoare : sirul () () se transformă în (()) ;

() – se obține un sir cu un cost mai mic cu 2. Orice astfel de caz este redus la situația următoare : sirul (()) se transformă în () ().

Rezolvarea se face asemănător cu punctul a) : se parcurge sirul inițial construind stiva în același mod și se contorizează de câte ori se întâlnesc paranteze pereche de cost 1.

Mai există un caz particular în care sirul de la stânga perechii () este parantezat corect, caz în care *swap-ul* este invalid deoarece nu se obține o parantezare corectă. Exemplu pentru parantezarea (()) (), interschimbând parantezele de pe pozițiile 5 și 6 se obține sirul invalid (()) (. Pentru a evita situația prezentată se verifică dacă stiva nu este vidă.

10. Valet

O configurație este identificată prin poziția mașinii pe care dorim să o scoatem din parcare (cx, cy) și poziția locului liber (hx, hy).

Pentru simplitate (și economie de memorie) vom codifica configurația printr-un număr întreg în care ultimele două cifre sunt cx , următoarele două cy , următoarele două hx și primele două hy . Vom rezolva problema efectuând un Lee pe configurații :

1. Inițializăm coada cu configurația inițială ($cx, cy, 1, 1$)
2. Cât timp coada nu este vidă și mașina nu a ajuns la ieșire :
 - extragem un element din coadă
 - analizăm succesiv vecinii spațiului liber pe care se află mașini parcate (adică poziții vecine pe care nu sunt stâlpi) și generăm noile configurații, mutând mașinile învecinate în spațiul liber; pentru fiecare configurație generată reținem numărul de mutări efectuate pentru a ajunge în configurația respectivă, apoi introducem configurația în coadă.

11. Predecesor

Soluția optimă va utiliza o stivă st de lungime n și un vector viz , tot de lungime n ($viz[k] = 1$ dacă elementul de pe poziția k din stivă are succesor sau 0 dacă nu are). Observăm că, dacă la un moment dat avem în sir o subsecvență de forma

$$a[i] > a[i+1] > \dots > a[j],$$

atunci este evident că un element $a[k]$ ($j < k$ și $a[j] < a[k]$) are ca predecesor pe $a[j]$, deci ceilalți termeni ai subsecvenței nu mai pot avea predecesori.

Vom introduce succesiv în stivă elementele sirului. La fiecare pas, în stivă elementele vor fi păstrate în ordine strict crescătoare. Când citim un element x din sir, avem cazurile :

1. x este mai mare decât elementul din vârful stivei. În acest caz, inserăm pe x în stivă.
2. x este mai mic decât vârful stivei. Atunci vom extrage din stivă toate elementele stivei strict mai mari decât x , iar cele marcate în viz cu 0 le contorizăm ca fiind fără predecesori. Apoi,

- a. dacă stiva e vidă, îl punem pe x în stivă nemarcat (nu are predecesori, încă!);
- b. dacă stiva nu e vidă, elementul din vârf îl marcăm cu 1 (are predecesor pe x) și apoi îl adăugăm pe x nemarcat în stivă.

La sfârșit, golin stiva pentru a contoriza și elementele rămase în stivă nevizitate.

Deoarece fiecare element din sir este introdus o singură dată în stivă și o singură dată este extras, complexitatea algoritmului este $O(n)$.

12. Cezar

Se poate utiliza metoda *Greedy*. Se elimină succesiv, dintre frunzele existente la un moment dat, frunza de cost minim. Toate nodurile au costul inițial 1. La eliminarea unei frunze, se incrementează cu 1 costul tatălui acesteia. Validitatea metodei rezultă din observația că, la eliminarea unei frunze oarecare, tatăl acesteia poate deveni frunză la rândul lui, dar cu un cost strict mai mare decât al frunzei eliminate. Frunzele se pot organiza ca un *min-heap*, structură care se actualizează în timp logaritmic.

7. Containere asociative

1. a. -301510 ; b. -3011510 ;
2. d ; 3. map : ac ; multimap : c
4. a, b, c, d, e, f, g, h, i;
- 5.
- 200 13 8
- 50 1
- 20 3 300 1 33
- 2 5 0
- 6.
- Ana 1
- Ana 3

Ana 5
 Dan 6
 Dan 7
 Dan 8
 Ion 4
 Xena 2
 Ana 1
 Dan 6
 Ion 4
 Ion 4

8. Stones

Metoda este *Greedy*: la fiecare pas se reunesc cele mai mici grămezi. Organizați grămezile ca un multiset.

9. Pikachu

Vom analiza cazul în care N este egal cu K . În această situație răspunsul optim se obține aducând elementele la valoarea medianei (definim mediana unui sir cu N elemente ca fiind al $N/2$ -lea element în ordine crescătoare). Aceasta este o afirmație cunoscută, demonstrația poate fi găsită în cartea *Introducere în algoritmi* de Cormen, Leiserson și Rivest.

Pentru cazurile în care K este mai mic decât N , observăm că nu are rost să considerăm subsecvențe de lungime mai mare decât K . Pentru subsecvența care începe pe prima poziție putem calcula ușor mediana și costul aducerii celorlalte elemente la această valoare în complexitate $O(K \log K)$. În continuare vom procesa următoarele subsecvențe în ordine, încercând să recalculem mediana și costul în mod eficient. Costul cerut este egal cu $(K/2) * \text{mediana} - \text{suma elementelor mai mici ca mediana} + \text{suma elementelor mai mari ca mediana} - (K-K/2-1) * \text{mediana}$. Observăm că două subsecvențe consecutive diferă prin doar două elemente. De aici tragem concluzia că avem nevoie de o structură de date care să suporte în mod eficient următoarele operații: inserarea unui element, ștergerea unui element, găsirea medianei.

Pentru aceste operații putem folosi un arbore de intervale sau un arbore echilibrat (este suficient un set din *STL*). Costul se poate recalculate la fiecare pas în timp constant.

10. Submatrix

Vom calcula pentru fiecare poziție latura maximă a submatricei pătratice care are colțul din dreapta jos în poziția respectivă și care conține maxim k numere distincte. Vom reține aceste valori într-o matrice *best*. Observăm că $\text{best}[i][j] \leq \text{best}[i-1][j-1] + 1$. Vom calcula valorile matricei *best* pentru fiecare diagonală independent. În momentul în care dorim să trecem dintr-o stare (i, j) în starea $(i+1, j+1)$, vom încerca să extindem pătratul cu colțul în (i, j) cu o unitate. Dacă noua submatrice va conține mai mult de k numere distincte, o vom micșora atât cât este necesar – micșorarea se face păstrând colțul din dreapta jos în $(i+1, j+1)$. Pentru a putea efectua rapid operațiile descrise, vom normaliza valorile din matrice și ne vom menține un vector de frecvențe. Această soluție are complexitatea $O(N^3)$, deoarece pentru fiecare diagonală vom parcurge fiecare element al matricei de maxim două ori.

Anexă – Caracteristici ale principalelor containere

	vector	deque	list	set	multiset	map	multimap
Organizare internă	Vector alocat dinamic	Vector de vectori	Listă dublu înțanjuită	Arbore binar echilibrat	Arbore binar echilibrat	Arbore binar echilibrat	Arbore binar echilibrat
Elemente	Valoare	Valoare	Valoare	Valoare	Valoare	Pereche cheie-valoare	Pereche cheie-valoare
Permite valori egale	Da	Da	Da	Nu	Da	Nu pentru cheie	Da
Permite acces aleator	Da	Da	Nu	Nu	Nu	Nu	Nu
Iteratori	Cu acces aleator	Cu acces aleator	Bidirectionali	Bidirectionali (Element constant)	Bidirectionali (Element constant)	Bidirectionali (Cheie constantă)	Bidirectionali (Cheie constantă)
Căutare element	O(n)	O(n)	O(n)	O(log n)	O(log n)	O(log n) pentru cheie	O(log n) pentru cheie
Însărcarea/extragerea elementelor	O(n)	O(n)	O(1) oriunde (O(1) doar la început și sfârșit)	O(log n) oriunde	O(log n) oriunde	O(log n) oriunde	O(log n) oriunde
Eliberare memorie la extragere element	Nu	Uneori	Da	Da	Da	Da	Da
Permite rezervarea memoriei	Da	Nu	–	–	–	–	–
Operator de indexare	O(1)	O(1)	–	–	–	O(log n)	–

Bibliografie

- Andonie, Răzvan ; Gârbacea, Ilie, *Algoritmi fundamentali. O perspectivă C++*, Editura Libris, Cluj Napoca, 1995.
- Cerchez, Emanuela ; Şerban, Marinel, *Programarea în limbajul C/C++ pentru liceu*, vol. I-III, Editura Polirom, Iaşi, 2005-2006.
- Farrell, Joyce, *Object Oriented Programming using C++*, Course Technology, 2009.
- Gălățan, Constantin, *C++. Introducere în Standard Template Library*, Editura All, Bucureşti, 2008.
- Iorga, Valeriu ; Opincaru, Cristian ; Stratan, Corina ; Chiriţă, Alexandru, *Structuri de date și algoritmi. Aplicații în C++ folosind STL*, Editura Polirom, Iaşi, 2005.
- Jamsa, Kris ; Klander, Lars, *Total despre C și C++*, Editura Teora, Bucureşti, 2001.
- Josuttis, Nicolai, *The C++ standard library. A tutorial and reference*, Addison Wesley, 1999.
- Meyers, Scott, *Effective STL. 50 Specific Ways to Improve Your Use of the Standard Template Library*, Addison Wesley, 2001.
- Musser, David ; Derge, Gillmer ; Saini, Atul, *C++ programing with Standard Template Library*, Addison Wesley, 1996.
- Sintes, Tony, *Teach yourself Object Oriented Programming in 21 days*, SAMS Publishing, 2002.
- Stroustrup, Bjarne, *What is “Object Oriented Programming”?* (1991 revised version).
- Stroustrup, Bjarne, *Programming Principles and practice using C++*, Addison Wesley, 2009.
- Standard C++ Library reference*, <http://www.cplusplus.com/reference>.
- Standard Template Library Programmer's Guide*, <http://www.sgi.com/tech/stl>.

LA EDITURA POLIROM

au apărut :

- Emanuela Cerchez, Marinel Șerban – *Programarea în C/C++ pentru liceu* (vol. I)
Emanuela Cerchez, Marinel Șerban – *Programarea în C/C++ pentru liceu. Metode și tehnici de programare* (vol. II)
Emanuela Cerchez, Marinel Șerban – *Programarea în C/C++ pentru liceu* (vol. III)
Emanuela Cerchez, Marinel Șerban – *PC. Pas cu pas*
Silvia Curteanu – *PC. Ghid de utilizare*
Ştefan Tanasă, Cristian Olaru, Ştefan Andrei – *JAVA de la 0 la expert*
Dragoș Acostăchioae – *Administrarea și configurarea sistemelor Linux* (ed. a III-a)
Dragoș Acostăchioae, Ovidiu Ene – *FreeBSD. Utilizare. Administrare. Configurare*
Ştefan Trăusan-Matu – *Programare în Lisp. Inteligență artificială și web semantic*
Marin Fotache – *Proiectarea bazelor de date. Normalizare și postnormalizare.*
Implementări SQL și Oracle
Valeriu Iorga, Cristian Opincaru, Corina Stratan, Alexandru Chiriță – *Structuri de date și algoritmi. Aplicații în C++ folosind STL*
Silvia Curteanu – *PC. Elemente de bază și utilizare*
Sabin Buraga – *Programarea în Web 2.0*
Doina Logofătu – *Algoritmi fundamentali în Java. Aplicații*
Silvia Curteanu – *PC. Elemente de bază și utilizare*
Silvia Curteanu – *Inițiere în Matlab*
Mircea Băduț – *Sisteme geoinformatiche (GIS) pentru electroenergetică*
Dorel Lucanu, Mitică Craus – *Proiectarea algoritmilor*
Marin Fotache – *SQL. Dialecte DB2, Oracle, PostgreSQL și SQL Server*
Ana Întuneric, Cristina Sichim, Daniela Tarasă – *Aplicații Windows în Visual C# 2008 Express Edition. Aplicații cu baze de date SQL Server 2008*
Mircea Cezar Preda (coord.), Ana-Maria Mirea, Doina Lavinia Preda, Constantin Teodorescu-Mihai – *Introducere în programarea orientată-obiect. Concepte fundamentale din perspectiva ingineriei software*
Cosmin Vârlan – *ActionScript 3.0. Programare Web în Flex și Flash*
Mircea Băduț – *AutoCAD-ul în trei timpi. Inițiere, utilizare, performanță* (ed. a III-a)
Traian Anghel – *Tot ce trebuie să știi despre Internet*
Mircea Băduț – *Calculatorul în trei timpi* (ed. a IV-a)
Emanuela Cerchez, Marinel Șerban – *Programarea în C/C++ pentru liceu. Programare orientată pe obiecte și programare generică cu STL* (vol. IV)

www.polirom.ro

Redactor : Adrian Botez
Tehnoredactor : Irina Lăcătușu

Bun de tipar : octombrie 2013. Apărut : 2013
Editura Polirom, 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@polirom.ro
București, Splaiul Unirii nr. 6, bl. B3A, sc. 1, et. 1,
sector 4, 040031, O.P. 53 • C.P. 15-728
Tel. : (021) 313.89.78 ; E-mail : office.bucuresti@polirom.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 : office@luminatipo.com
