Предговор

Настоящото издание е учебник, създаден по едноименния курс за студентите от базова програма “Информатика и телекомуникации” на Нов Български Университет. Това предопределя неговата задача да даде от една страна основни познания, необходими като първа стъпка в по-нататъшното обучение на бъдещите информатици, а от друга – достатъчен обем от основни знания и умения на бъдещите бакалаври по телекомуникации, които няма да изучават други курсове, свързани с алгоритмите.

При разработването на методическия подход на представения тук материал, първата цел беше той да бъде разбираем, общообразователен, леко да се запаметява и да създава стабилни представи за основни понятия в алгоритмите и програмирането. Поради това материалът е придружен с много аналогии, метафорични илюстрации, опорни схеми и т.н.

Втората цел беше да подпомогне обучаемите да създават сами програми за по-елементарни алгоритми, като им даде методическа опора и тренира у тях умението да извършват преход от задачата към алгоритъма и към съответстващ му програмен текст. Поради това изложението е придружено с много подробни обяснения както на методите и алгоритмите, така и на практически задачи “трениращи” алгоритмичното мислене и уменията за кодиране.

Третата цел беше знанията и уменията да бъдат универсализирани по отношение на използвания език за програмиране. Това наложи да се изобрети един подход за обобщено изобразяване на алгоритъма-програма посредством схема за управление, която позволява “пренасяне” в който и да е процедурен език за програмиране. Като опорни за примерите са избрани Pascal, като идеен основоположник на структурното програмиране, най-близък до естествения език и най-отчетлив на абстрактно равнище на организация на работата с паметта и данните, и C, който е основа за привикване към синтаксиса на най-често използвания език при обучение в обектно-ориентирано програмиране. Практически няма никакво значение за обучаемите кой от езиците ще изберат за упражненията в компютърните класове, дори се препоръчва да опитат и двата опорни езика.

Предполага се, че студентите са програмирали в средния курс на обучение и са запознати на начално равнище с базовия синтаксис на някой език за програмиране, както и със значението на термини от рода на “машинна памет”, “компилатор”, “бит”, “взаимно еднозначно съответствие”, “ред”, “вектор” и други базови термини от областта на информатиката и математиката.

Учебникът съставен от две функционални части както следва:

Първа част – основно изложение. Основните знания и приложените примери в нея са разделени на три дяла като материал за аудиторни занятия. Автор на тези дялове е Велина Славова.

Втора част – решени примери на задачите за самостоятелна работа, съдържа и задачи с методическо значение за умението за съставяне на алгоритми при тясно обвързване с примерите от теоретичната част. Приведени са и работещи

програмни текстове, с коментари, предназначени за упражнения с компютър и разширяващи познанията по програмиране. Този дял от учебника е разработен от Станислав Иванов.

Благодарности

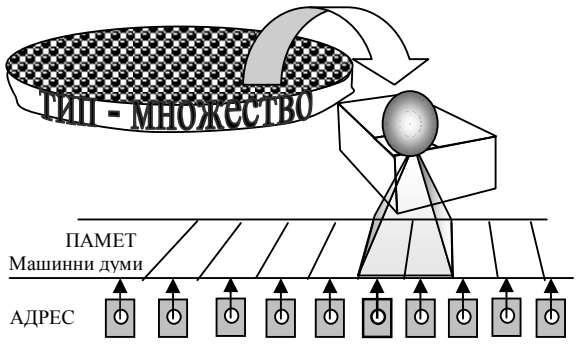
Методическият подход тук се базира на теорията, разработена и експериментално доказана от професор Лоуранс Барсалоу (prof. Lawrence W. Barsalou), свързана с откриване на наличието на така нареченото “базово ниво” на концептите и на параметри, свързани с основни характеристики на вътрешното структуриране в човешката понятийна система. Авторът на това не съвсем типично изложение на учебен материал от областта на точните академични дисциплини, Велина Славова, изказва благодарност на Lawrence W. Barsalou, който се запозна с ръкописа на лекциите и насърчи настоятелно издаването на учебно помагало по записките. Авторите изказват благодарност на студента Добромир Динев, който набра първи вариант на текста по ръкописа.

Структуриране на данните

В настоящото учебно помагало разглеждаме алгоритъма като единство на три съставни части – среда, действия и управление, обособени условно заради специфичната роля на всяка от тях на абстрактно и на машинно равнище. Най-общо, средата е машинната памет, организирана по определени правила за работата на алгоритъма, действията са изпълнимите от машината команди, довеждащи до промени на записаното в паметта, а управлението е тази съставка, която "организира" действията над средата така, че да бъде решавана конкретна задача. Трите условни съставки образуват алгоритъма, който изпълнява предназначението си благодарение на тяхното взаимодействие. Очевидно при съставяне на алгоритъм тези съставки се замислят паралелно и зависят една от друга.

Средата е най-силно обвързана с машината, защото пряко свързва работата на алгоритъма с машинната памет. Организирането на среда за работата на един алгоритъм съдържа и елементи на формализъм, на част от които се спряхме. Нека припомним накратко парадигмата, в която развихме началната представа за "среда". Образно представихме променливата като празна кутия, в която може да бъде поставено съдържание, т.е. стойност на променливата. Кутията има име и тип. Името се използва пряко, за да се избере кутията, с чието съдържание ще се извършват действия. Типът на променливата показва какво би могло да бъде съдържанието и какви действията са възможни с него. Работихме с представяната в машината "базова" среда, а именно – стандартните типове.

Както е илюстрирано, "*типът на променливата*" дава основната връзка между "множество с операции" и реализацията на машинно равнище. За стандартните типове това става с кодиране по определени правила, като елементите на абстрактното множество се представят чрез поредици от битове.



Паметта е организирана като вектор от машинни думи, а всяка дума има адрес, осигуряващ на алгоритъма достъп до записаното в думата. Адресите са цели положителни числа, с които думите са "номерирани".

Средата на алгоритъма може да бъде изграждана не само от "прости" променливи-кутии. Тя може да съдържа и по-сложни елементи, "конструирани" от програмиста по определени правила. Такива "конструкции" в средата, изразяващи някаква по-сложна организация на данните, с които алгоритъмът работи, се наричат "*структури от данни*".

Както при съставянето на собствени блокчета за управление, конструирането на по-сложни елементи на средата става с използване на съществуващия "градивен материал" – променливи-кутии с дефиниран тип на съдържанието.

Нека си представим, както е илюстрирано вдясно, че за работата на алгоритъма е необходима една "по-сложна кутия", която да се третира от алгоритъма като единно цяло. Тя може да бъде изградена чрез някакво организирано съчетаване на "прости кутии". Както всеки елемент на средата, така и илюстрираната "съставна" кутия има име. Използването на това име в текста на алгоритъма означава обръщение към съдържанието на този "многокомпонентен състав".

Да разделим на две нива това, което представяме чрез образа "кутия". Едното ниво е свързано представяното множество от стойности, а другото – със съответствието на съдържанието на една съставна променлива с адресите и думите в паметта.

1.1. Нестандартен тип на променливата

Множеството, от което разгледаната в горния пример променлива може да получава съдържания (стойности) се нарича **тип** на променливата.

Очевидно това множество няма предварително подготвена в езика кодировка и именно поради това носи името "нестандартен тип". Нестандартните типове се конструират и "обявяват" от програмиста.

Програмно, дефинирането на нестандартен тип става със "заявка за нестандартен тип" – така наречения ***конструктор*** на типа. Конструкторът има задачата да "обяви" типа, като му даде *име* и *опише* по определени правила какво представляват елементите на новото множество от стойности.

Заявките за нестандартни типове се разполагат в програмния текст в частта "заявки", като предшестват тези за имена и типове на променливите. След като типът е обявен и описан, необходимите за работата на алгоритъма променливи от този тип се декларират със съответни заявки. Примерът, илюстриран на фигурата, съответства на следните заявки:

Тип Mytype = описание;

Променливи M, G – от тип Mytype;

Характерно за работата с променливите от един и същи тип е, че стойностите им могат "взаимно" да се присвояват. В нашия пример, операторът:

$M \leftarrow G$;

присвоява на променливата M стойността, записана в променливата G, каквото и да означава това. Да видим какво би могло да означава това, като разгледаме другото ниво на образа "кутия", а именно – съдържанието и начина на разполагането му в паметта.

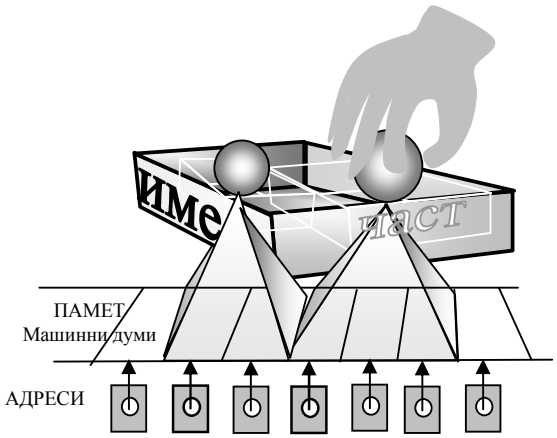
В Pascal конструкторът на тип има следния общ вид:

Type <име на типа> = < описание на типа по някакво правило>

- ***Достъп до съдържанието на променливи-структури***

На илюстрацията долу е показано образно съдържанието на съставна променлива, представено в парадигмата "кутия-съдържание".

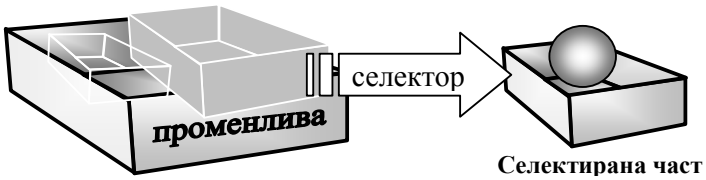
Както е показано, съдържанието се състои от обособени части, а именно стойностите на всяка от под-кутиите. Алгоритъмът може да работи само с данни от "известен му тип", записани на "известно му" място в паметта. Ако съдържанията на под-кутиите на илюстрираната променлива са от стандартен тип и адресите, на който те са записани са достъпни, т.е. те са "известни" на алгоритъма, то той може да работи с тях.



Тук ще се спрем на такива структури, в който за да "се достигне" до някоя записана в паметта стойност не се налага да се управлява "явно" работата на машината с адресите. Ще разгледаме организирането на такава среда, която позволява да се работи само с имената на "кутиите". Адресите остават "скрити"

за програмиста, а определянето им става автоматично при изпълнението на програмата. Дали адресът се определя при компилиране, изчислява се впоследствие по определено правило или пък се пресмята при всяко стартиране на процедура, няма значение. Важното е, че за да се работи със съдържанието на всяка под-кутия, е достатъчно тя да бъде "назована". Такъв вид структурирани елементи от средата на алгоритъма, достъпът до частите на които е възможен непосредствено "през името" се наричат "*структури с пряк достъп*".

Работата над съдържанията, записани в такива структури може да става само след посочване на точната "под-кутия", с която ще се работи. т. е. след нейното *селектиране.*



До всяка "под-кутия" се достига чрез посочване на името на съставната променлива и *селектиране на самата подутия* В езика за програмиране това става със специално средство за изказ, наречено *селектор.*

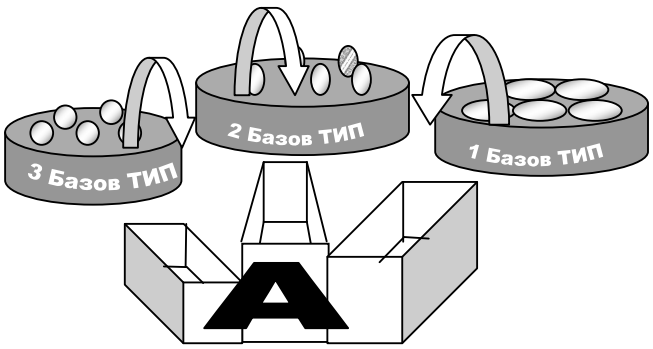
Независимо от конкретния синтаксис, семантиката на селектора е следната:

Име на съставната променлива	разграничител	**Идентификатор на частта, с чието съдържание ще се работи**

Резултатът от селектирането е една "кутия", която има определен собствен тип.

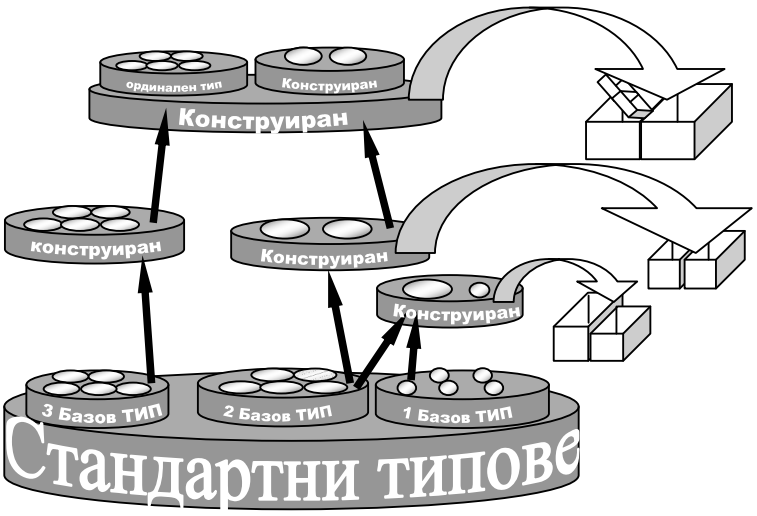
На рисунката долу е илюстрирана променлива с име "A", състояща се от три под-части. Съдържанието на отделните детерминирани части на променливата A могат да бъдат само от дефинирани в средата на дадения алгоритъм типове. За всяка такава детерминирана част от A се казва, *че тя е от базов тип X.*

Базовият тип определя операциите, които могат да се извършват над съдържанието на конкретната част от променливата.



Ако всички детерминирани части в една структура са с еднакъв базов тип, структурата се нарича *хомогенна*. Ако те не са с еднакъв базов тип, тя е *нехомогенна*.

Базовият тип не е непременно стандартен тип. Той може да бъде всеки допустим и обявен по правилата тип, включително нестандартен. Това позволява в средата да се организират конструкции с повече "нива на влагане" по принципна схема, илюстрирана на фигурата долу:

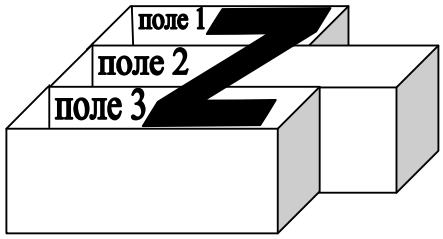


- Основните градивни елементи на типовете са стандартните типове**,** дори да не участват *пряко*, в някой от конструираните типове.
- За базов тип на конструирания тип може да бъде "вграден" друг конструиран тип. "Влагането" може да бъде на много нива (на схемата са изобразени само две нива).
- Прилагането на повече нива на вложение налага да се прилага многократно селектиране, за работи със стойностите (селектор от селектор от селектор…).

1.1.1 Структурата "Запис"

Записът е нехомогенна структура с пряк достъп. Долу е илюстрирана една променлива Z, която е от тип ***запис***.

Променливата се състои от няколко (в случая 3) "под-променливи", наречени *полета* на променливата-запис. Всяка "под променлива" има собствен ***базов тип***. Полетата са неразделна част на променливата-запис и за да се работи със съдържанието на някое от тях, то трябва да бъде селектирано от променливата-запис.

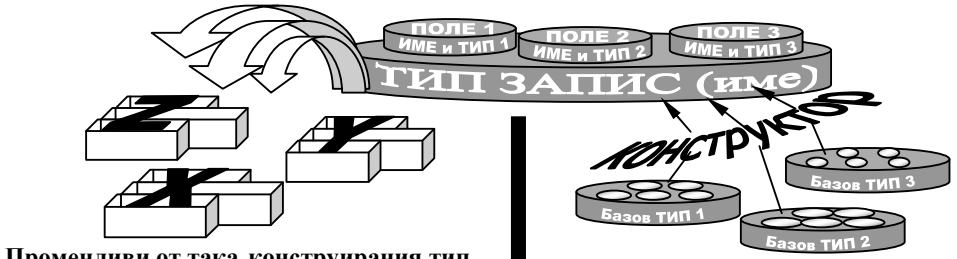


Самата променлива-запис има свой тип – множество, което има ***име***. Конструирането на типа за променливи-записи се прави, като се дефинират полетата, а именно – *на всяко поле се задава име и се посочва неговият тип.*

В Pascal например конструкторът на типа "запис" има следния синтаксис:

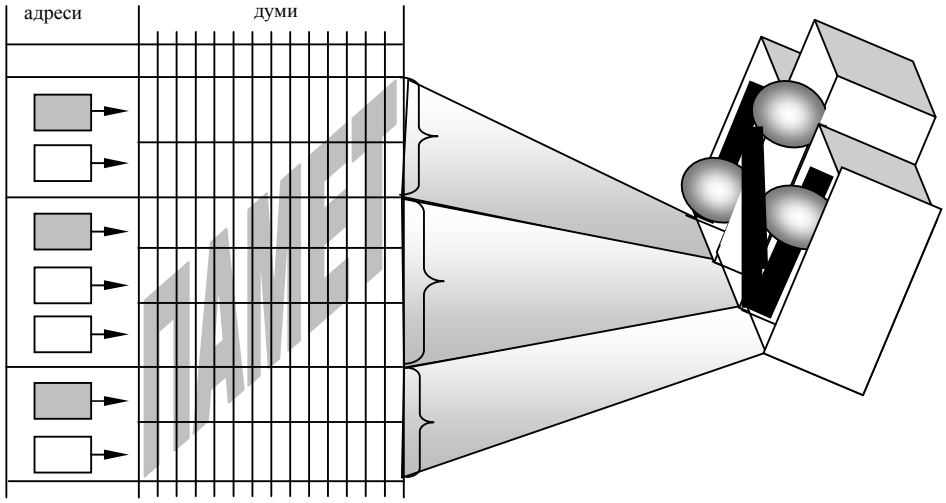
Type CARS = ***record***	***дефиниране*** на тип запис : име на множеството
Number : byte;	име и тип на поле 1
Model : string;	име и тип на поле 2
Costs : real;	име и тип на поле 3
End;	***Край на дефиницията***

След като типът е дефиниран с конструктора, могат да бъдат заявявани променливи от новосъздадения тип. Както обичайно, това става със заявка за име и тип на променлива. На илюстрацията долу е показано схематично действието на заявката за конструиране на тип и на заявката за обявяване на променливи от този тип.

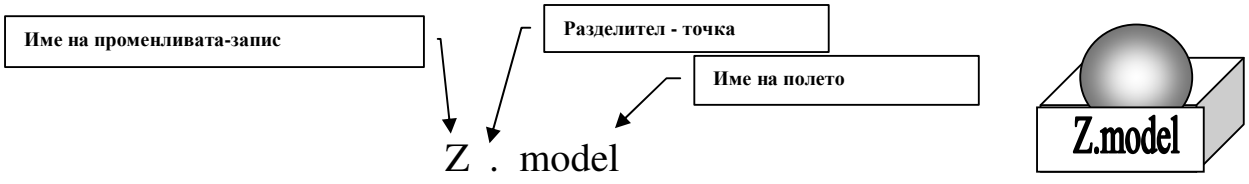


Променливи от така конструирания тип **Базови типове на полетата**

Принципно, вследствие на заявката за променливи от такъв тип, на всяка променлива се отрежда адрес и обем памет (брой думи). При типа запис *за всяко поле* на една променлива запис се отрежда *адрес и памет* (за съдържанието на "под-кутията" се запазват толкова машинни думи, колкото са необходими за съответния базов тип)..

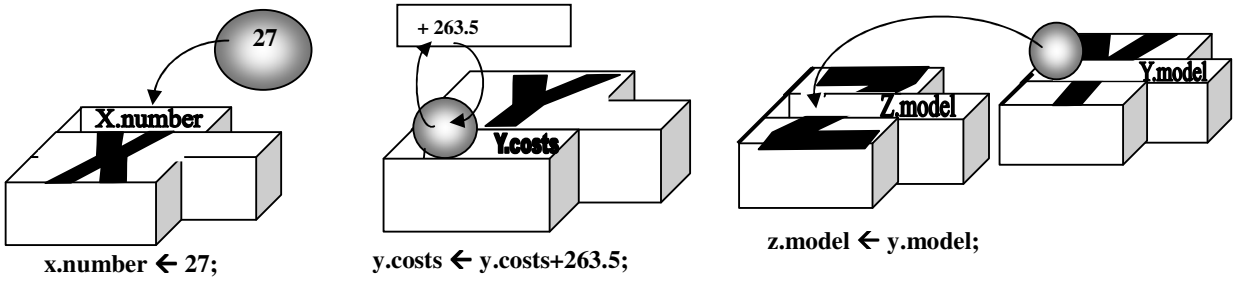


Илюстрираната на фигурата горе организация на отреждането на памет за полетата на променлива-запис показва как се осигурява прекият достъп до съдържанията. Първото поле от записа е разположено на адреса на самия запис, а всяко следващо поле – непосредствено след предшестващото го. Спазването на това правило позволява на компилатора да изчисли адреса на всяко селектирано поле. Ролята на селектора е само да посочи за кое именно поле от променливата-запис става въпрос, т.е. за коя "под променлива". Това напълно отговаря на работа с отделна променлива. Синтаксисът на селектора е:



Всъщност, фактът, че всяко поле от една променлива-запис има име, обявено при конструирането на типа, води до това, че самото поле има свое име, съставено чрез селектора.

Наличието на имена на полетата в рамките на записа позволява с полятата да се работи както с отделни променливи, както е илюстрирано с няколко примера долу.



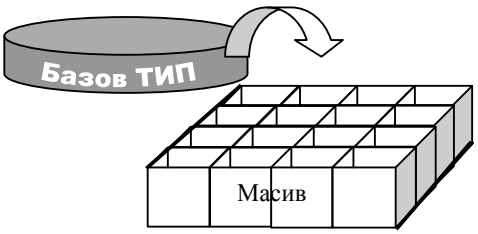
Основното практическо предимство на типа "запис" е неговата нехомогенност. От изложеното дотук става интуитивно ясно, че типът *запис* може удачно да служи за описание на различни свойства на един обект. Тези различни свойства в общия случай са от различни типове. Съчетаването на различни типове за описание на един обект е характерно за *базите данни*. Те използват таблици от вида:

променлива	Number	Model	Costs
X	27	BMW	381.4
Y	32	Renault	486.5
Z	43	Moskvitch	217.7

Освен за бази данни, типът "запис" е основно средство за организиране на *динамични структури* от данни. Използването му позволява да се поддържат удачно структури, чиито брой на данните не е предварително фиксиран. В тези случаи едно (или повече от едно) от полетата на променливата-запис е за данни, а друго (или други) е за *адрес(и)* на продължението на структурата.

1.1.2 Структурата "Масив"

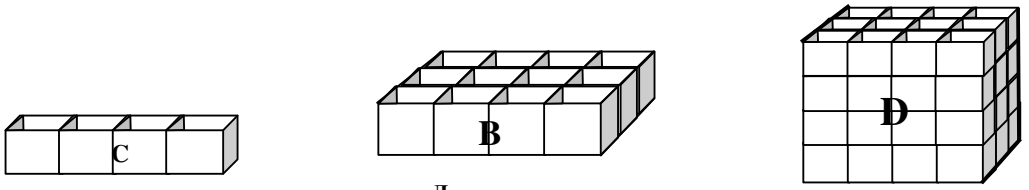
Масивът е хомогенна структура с пряк достъп. Исторически погледнато, в езиците за програмиране най-напред се е появила тази структура като средство за организиране на множество данни с еднотипно съдържание.



Всяка от детерминираните части на структурата има съдържание, което е от същия тип като това на всички останали "части".

Всички елементи – "клетки" на такава структура, са променливи от един и същи тип и всички действия, които могат да бъдат извършени над съдържанието на един, могат да се извършват и над всеки друг елемент. Същевременно това е структура с пряк достъп и всеки неин елемент се идентифицира с "името" си.

Правилото за идентифициране на елемент от масив в повечето езици за програмиране е заимствано от линейната алгебра, където "номерирането" на елементите на обекти от типа на вектори и на матрици става с посочване на позицията им долу вдясно (напр. A_{ij}) и се нарича *индекс*. По аналогия с линейната алгебра, където обектите са разположени в пространства, и масивите имат "*мерност*". Мерността и индексацията дават точно правило за това как да бъде идентифициран един конкретен елемент на масива. Всеки елемент-клетка на един n-мерен масив се идентифицира с *наредена n-орка* от индекси. Елементът има толкова индекси, колкото е "размерността" на пространството. Долу са илюстрирани "конгломерати" от еднотипни кутии, организирани като променливи-масиви:



Едномерен **Двумерен** **Тримерен**

Нека на горните илюстрации позициите по всяко направление са "номерирани" от 1 до колкото е броят на елементите по това направление. Индексите по всяко направление се разглеждат като отделни множества, наричани "*индексни множества*". Например, един тримерен масив има три индексни множества – по едно за всяко от направленията. Броят на елементите на структура като някоя от илюстрираните горе се пресмята, като се умножат мощностите на индексните множества. Полученият брой се нарича *мощност на индекса* на масива.

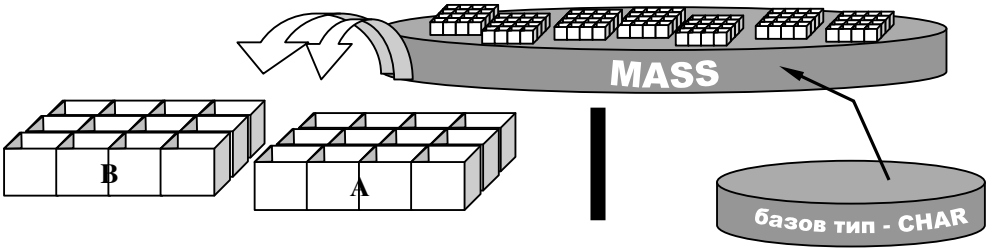
Индексирането в Pascal може да се извърши с множеството {1, 2, 3, ... , к}. Това множество е интервален ординален тип. Всички ординални типове са еквивалентни на такова множество. Следователно номерирането може да стане посредством елементите на *кой да е ординален*, наречен *тип на индекса* за съответното "направление" на масива. Всяко от индексните множества на типа "масив" има тип и мощност, който се задават в заявката – конструктор на типа. В Pascal конструкторът на типа масив изглежда например така:

Type Mass = *array* [1..4 , 1..4] *of* Char

Име на типа **Индексни множества** **Име на базовия тип**

На следващата илюстрация е показано действието на заявката-конструктор на типа от горния пример и на заявка за две променливи от този тип.

Var A,B : Mass;



Самият тип Mass е множеството на всички възможни стойности на масиви, които отговарят на правилото, посочено в конструктора. На въпроса "***колко са те?"***, отговорът е *"много са"*. Мощността на типа "масив" може да се пресметне по формули от комбинаториката. Елементите на базовото множество се разполагат по местата на масива, могат да се повтарят, при това наредбата им е от значение. Следователно, трябва да се изрази броят на елементите на комбинаторна конфигурация "*с наредба и с повторение"*. Формулата е:

$$\textit{Мощност на типа} = (\textit{мощност на базовия тип})^{\textit{мощност на индекса}}$$

В Pascal и C променливите-масиви се декларират най-често пряко – задава се име на променливата и се посочват индексните множества. Например така:

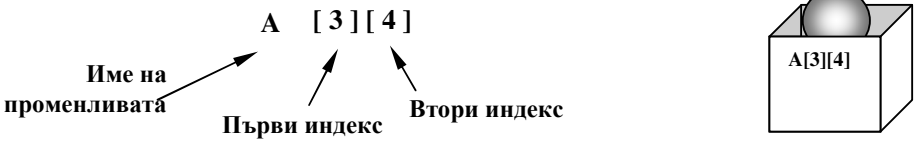
Var A : **array** [1..4,1..4] of Char ; **char** A [4] [4] ;

Var M : **array** [1..16,1..55] of Integer ; **int** M [16] [55] ;

Съществено е да се подчертае, че всяка променлива от типа "масив" *има точно определен брой елементи.* Това предопределя количеството памет, което се отрежда според заявката за променлива – масив.

Работата със стойностите на елементите на променливата-масив става след селектирането им.

Селекторът, приложен за примерния масив A от горната фигура изглежда например така:

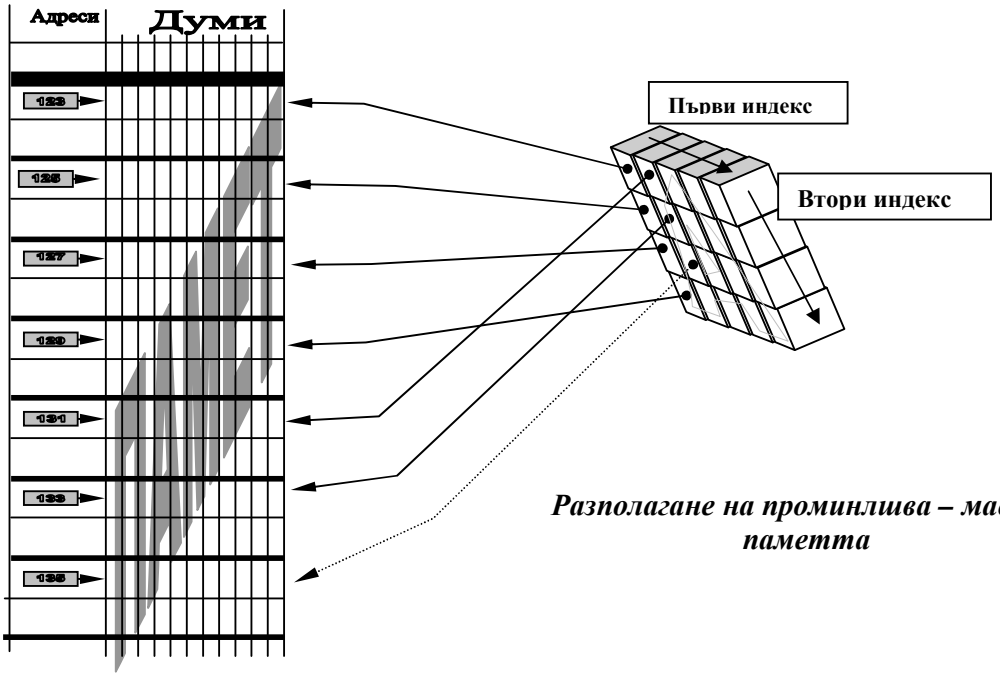


Както е илюстрирано горе, резултатът от селектирането е един елемент, с който може да се работи както с всяка друга променлива. "Името" на получената променлива се образува след "прилагане" на селектора.

Да дадем някои допълнителни обяснения за организирането на масива в паметта на машината, за да изясним по-подробно ролята на селектора. За компилатора стойностите на посочените в селектора индекси служат за изчисляване на адреса, на който се намира съдържанието на селектирания елемент.

- ***Реализация на прекия достъп до елементите на масив***

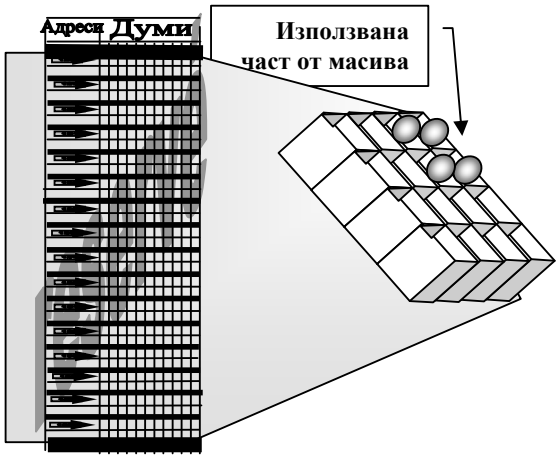
На следващата илюстрация е показан пример за това как се разполага в паметта един двумерен масив A, чийто базов тип изисква обем памет в размер на две машинни думи. Съществува *твърдо правило*, по което за съдържанията на елементите на масива се "запазва" точно определен адресиран сектор от паметта. Това правило зависи от езика за програмиране (ние не го "виждаме" докато програмата работи или докато я пишем). Например, в Pascal правилото е: най-напред са местата за всички елементи за които *последният* индекс се мени от началната до максималната си стойност, а всички останали индекси са на началната си стойност. После са местата на всички елементи за които пред-последният индекс е на втората си стойност, последният се мени от първа до последна, а всички останали са на начална и после се записват ... Да не се уморяваме с това. *Правилото е твърдо*. От това следва, *че за всеки елемент на масива има точно определено място на което се намира неговото съдържание*.



Разполагане на проминлшва – масив в паметта

Прекият достъп до стойностите на елементите, реализиран посредством пресмятане на индексите, посочени в селектора, е съществено практическо преимущество на структурата "масив". Ако в текста на програмата е записано например A[3][4], адресът на този елемент се изчислява в момента на компилиране (тогава се резервира паметта) и се "знае" през времето на цялото изпълнение на програмата. Ако в текста на програмата е записано A[i][j], където i и j са променливи, *адресът на* A[i][j] *се изчислява по време на изпълнение* **за** тези стойности на i и j, които са получени в момента. Такава организация позволява на равнището на алгоритъма с масива да се работи така, както това става например с матрица, като се използва широко разпространеният формализъм на индексирането.

Да обърнем внимание на факта, че по принцип независимо от това дали в една програма със стойността на променлива, заявена по описаните дотук начини, се "работи" или не, мястото й е "заето", пази се заедно с отредения му адрес до завършването на програмата или процедурата, и не може да бъде "давано" на друга променлива. Този вид отреждане на памет се нарича "*статично отреждане*".



Илюстрацията вдясно показва как може паметта, запазена за масив, да се използва по "полезен начин" само частично. Статичното отреждане на памет е съществения недостатък на структурата "масив" – то води до разхищение на машинна памет.

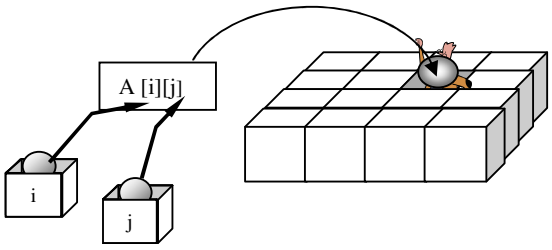
От изложеното дотук за "отреждането" на памет за елементите на масив следват два извода:

- Съдържанието на всеки елемент на масива се намира на точно определен, резервиран за този елемент адрес.
- Памет за целия масив е запазена и не е използваема за други цели, независимо от това дали се използва напълно или частично.

➢ *Управление, свързано с работата над масиви*

Над съдържанията на елементите на масива може да се работи за всеки елемент поотделно, както с прости променливи от дадения базов тип. За да се "идентифицира" елементът, трябва да бъдат посочени неговите *индекси*. Ако в текста на програмата е записано направо A[3][4], индексите са посочени и елементът е фиксиран. По-често индексите са *променливи* от съответния ординален тип. Долу е илюстрирано "идентифицирането" на елемент с индекси – стойностите на променливите i и j.

Очевидно A[i][j] може да бъде всеки елемент на масива в зависимост от стойността на променливите i и j. Ако стойностите на i и j са извън границите, посочени при описанието на масива, това ще доведе до опит за достъп до памет, непринадлежаща на масива.



Съществено е, че контрол за подобен неправомерен достъп не може да бъде осъществен по време на компилацията, защото индексите-променливи получават стойности едва по време на изпълнение на програмата. От всичко изложено дотук следва заключението, че управлението за работа над масив се свежда до управление на *индексите*.

- ***Обхождане на масив***

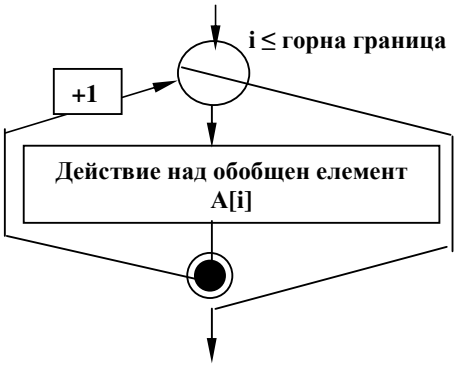
Обхождането е много често срещана алгоритмична операция, извършвана над съдържанието на елементите на структурите от данни. Да се *обходи* една структура означава да се посетят систематично и без прескачане всичките й съставни елементи, като над всеки елемент се извърши някакво действие D – едно и също за всички елементи. В частност, *обхождането на масив* означава извършване на някакво действие *поотделно и последователно над всеки елемент на масива*.

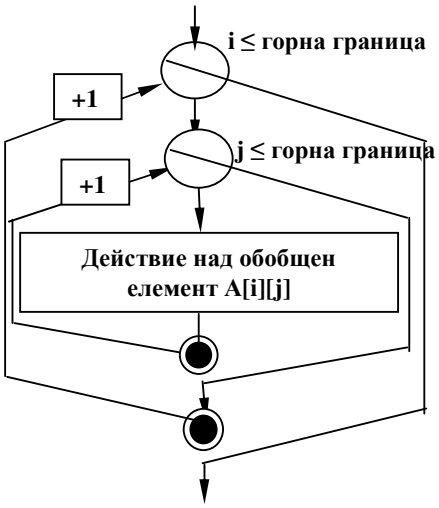
Като се има предвид, че "идентификаторът" на всеки елемент са индексите му, логично е да бъде приложено действието над един "*обобщен елемент*":

A [*индексите са променливи*].

"Обобщеният" елемент става *конкретен* при *конкретни* стойности на променливите-индекси. Изисква се само индексите да се менят по някакъв систематичен начин. Удобно е индексите да се менят в *цикъл*. Поради факта, че масивът има винаги индекси от ординален тип, цикълът може да бъде организиран по вграден брояч. Това е и видът управление, прилаган най-често над масив – *изменение на индекси с цикъл по вграден брояч.*

Долу е илюстрирано управлението при обхождане на едномерен и на двумерен масив. Очевидно обхождането на повече-мерен масив налага надстрояване на още цикли. Правилото е очевидно – колкото мерен е масивът, толкова вложени цикли се налагат за обхождането му.





Да обсъдим проблема със запазената за масива памет и фактически използваната памет от гледна точка на управлението на индексите. Най-често структурата *масив* се използва за моделиране на среда, в която има множество ***еднотипни данни***, но не е предварително известно *точно колко са те*. Тогава при избора на средата се "организират" масиви, които биха могли да се използват като *частично* запълнени с данни, както е показано на илюстрацията за резервираната за масива памет. Най-често такова разхищение на памет е допустимо в разумни граници. Все пак, не бива да се забравя, че то съществува и че в някои случаи се налага програмистът да подходи по-пестеливо към ресурса памет. Нашата задача в момента е да изясним въпроса с управлението

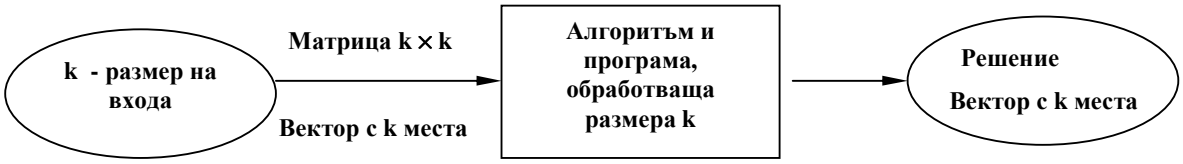
на индексите при "частично" запълване с данни на заявена променлива – масив. Да приведем за пример една позната задача.

Дадено:
коефициентите и свободният член в система линейни алгебрични уравнения,например:
$4x_1-2x_2+36x_3+ \ldots +43.5x_n=8$
$98x_1-12x_2+23x_3+ \ldots +26x_n=-112$
$63x_1-12x_2+69x_3+ \ldots +x_n=126$
…..
…..
$26x_1-43x_2+420x_3+\ldots+x_n=386.5$

Търси се:
Векторът x
Решението:
x_1 x_2 x_3 x_n

Задачата е да се състави алгоритъм и програма за решаване на система линейни алгебрични уравнения. Алгоритмизацията на решението изисква коефициентите пред n-те неизвестни в n-те уравнения да бъдат третирани като *матрица*. Матрица се моделира най-естествено с *двумерен масив*. Този масив съдържа коефициентите пред n-те неизвестни от n-те уравнения.

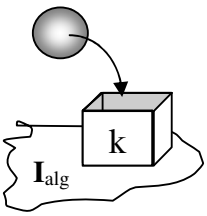
"Организирането" на коефициентите в двумерен масив изисква в конструктора на типа да се посочи *колко местен* е масивът. Ясно е, че става въпрос за моделиране на квадратна матрица $n \times n$. Но колко е n? Това, което обикновено се прави, е че се "запазва" едно конкретно n. Например 50. То ще бъде изцяло запълнено само при решаване на система от 50 уравнения с 50 неизвестни. Тогава средата на алгоритъма ще позволява да се реши всяка система от k уравнения с k неизвестни, при условие, че $k \leq n$.

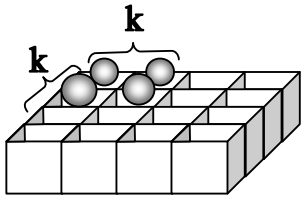


Ще наричаме k *размер на входа*, или още – размер на задачата.

Може да се каже, че средата на алгоритъма е "конструирана" да решава задачата за размери на входа от 1 до 50. Паметта, резервирана за такава програма ще бъде винаги като за размер на входа 50.

Размерът на задачата за конкретното изпълнение на програмата се задава от потребителя в отговор на въпроса: "колко уравнения със също толкова неизвестни има?" отговор: k – едно цяло положително число, по-малко или равно на 50.






```
For i:=1 to k do
Begin
    Действия
    …
End;
```

```
for (i=0; i< k, i++)
{
    Действия
    …
}
```

Всички следващи действия трябва да са обвързани с променливата К, която ще изпълнява функция на *горна граница* на циклите по брояч, както е илюстрирано на примера горе.

1.1.3 Алгоритъм за намиране на максимален (минимален) елемент на масив

Една от най-често срещаните задачи е тази за намиране на минималната или максималната стойност, записана в някой от елементите на масив. Тъй като обикновено всеки две стойности на елементи са сравними помежду си, да се търси максимална стойност е все едно да се търси най-голямата стойност, записана в масива. Тази задача, освен за прякото ползване на резултата като "сведение за състоянието" на данните, представлява добре дефинирана *подзадача*, която се използва като *инструмент* за решаване на голям брой разнообразни алгоритмични задачи. Ще се запознаем с нея подробно на този начален етап, а в последствие ще я използваме като основен градивен блок на по-сложни алгоритми.

Нека, за да работим с конкретен пример, да фиксираме задачата като задача за намиране на *максимален елемент* и да подчертаем, че става въпрос за стойността, записана в елемента. "Обръщането" на задачата от такава за максимален в такава за минимален елемент изисква само в алгоритъма да се сменят условията α, по които очевидно ще се налага да се правят проверки.

Постановка на задачата:

Даден е:	**Търси се**:
Масив, чийто елементи имат стойности. (Не се интересуваме как тези стойности са се озовали в масива).	Коя е най-голямата стойност? Къде се намира тя, т.е. кой е елементът, в който тя е записана?

Основният проблем за разбирането на алгоритъма на тази проста задача произтича от факта, че *човек* много бързо и без никакви усилия намира посочените горе търсени величини при обозримо количество данни. Например:

1	3	2	0	-1	9	8	1	-8	4
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**

Съвършено ясно е, че най голямата стойност е 9 и е записана на 6-то място. Ясно е на … четящия човек.

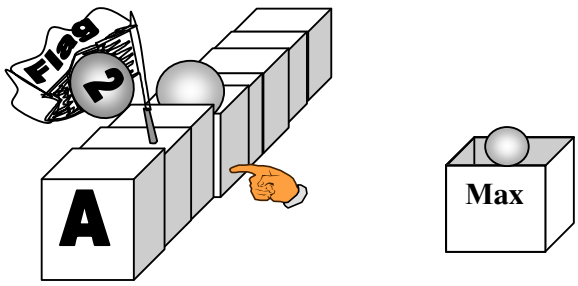
Алгоритъмът, обаче:

1. Не "вижда" всички данни едновременно, а "работи" с тях *една по една.*

2. Не "помни", освен в специално предоставена му тази цел променлива.

Задачата може да се реши чрез *обхождане*, както е илюстрирано долу. Да се ограничим с едномерен масив, имайки предвид, че обхождането е възможно за масиви с произволна "мерност".

Илюстрацията вдясно показва каква *среда* трябва да се осигури на алгоритъма и каква е принципната схема на работата му. Една специално създадена променлива Max има за задача да "съхрани" максималната от *стойностите*, записани в масива. Друга променлива – Flag – ще съхрани *индекса* на елемента от масива, който е с максимална стойност. Очевидно типът на Max е както този на елементите на масива, а този на Flag е като типа на индексите на масива.



Масивът се обхожда със сравняване и евентуална промяна на стойности. Сравняват се един по един елементите от масива със стойността, записана в променливата Max. В началния момент променливата Max има стойността на първия елемент от масива, а по-нататък, "обхождайки", за всеки елемент се прави проверка: "*е ли стойността на елемента по-голяма от тази, записана в променливата Max*". Ако се окаже, че стойността на елемента от масива е по-голяма от текущата стойност на Max, Max приема нова текуща стойност – стойността на въпросния елемент от масива. След като обхождането приключи, в Max е останала записана максималната стойност на елемент от масива. Напълно аналогични са разсъжденията за Flag.

Ще съставим алгоритъма по Top-down методика.

Ниво нула –алгоритмична задача:

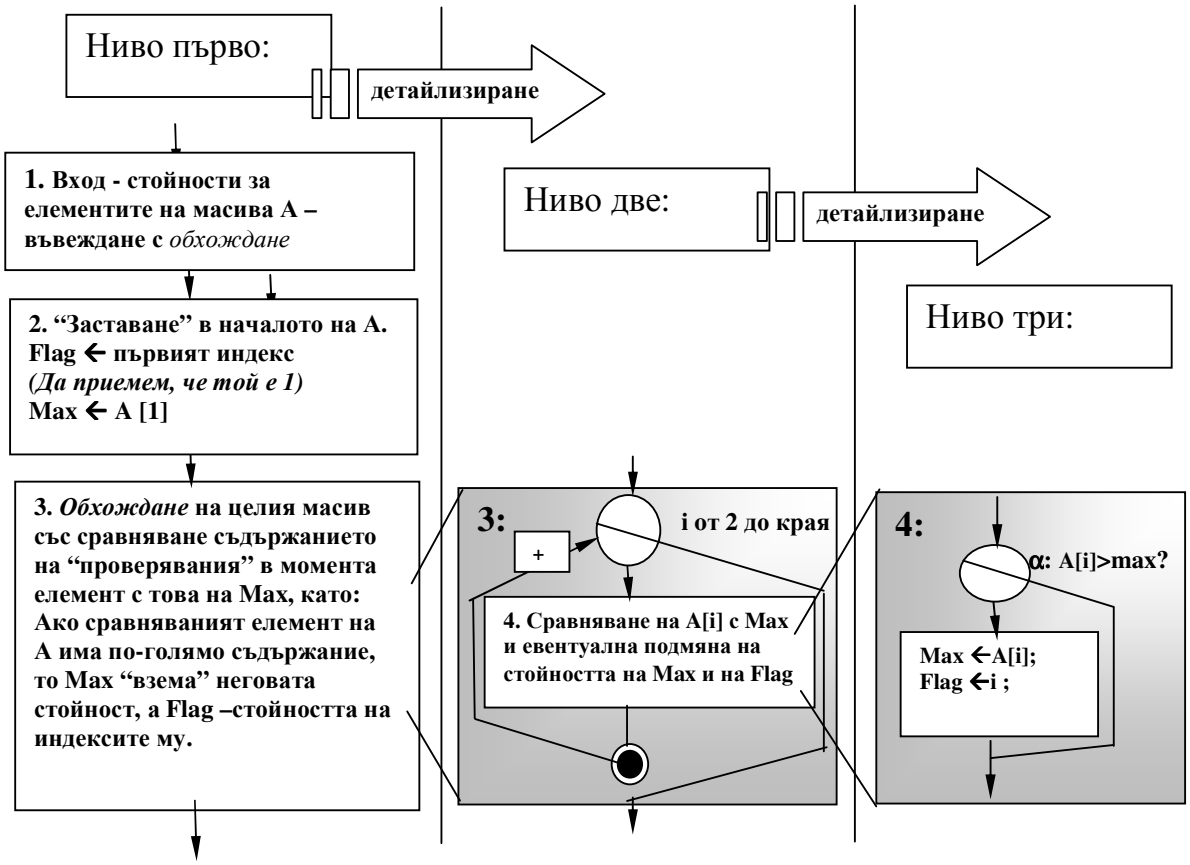
Променливи в средата:

i е номерът на текущия елемент при обхождането,

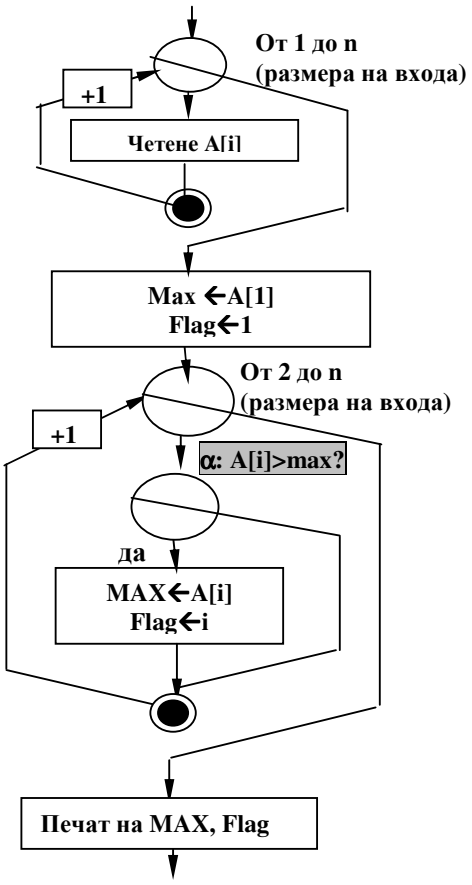
A[i] е стойността на посетения при обхождането елемент,

Max е стойността на намерения най-голям,

Flag е индексът, позицията на която Max е бил записан.



Този алгоритъм е много простичък и дори на "ниво първо" е почти програма. Полученият алгоритъм е показан на схемата вдясно, готов за "превод" на език за програмиране.



Да посветим няколко изречения на едно съществено понятие, а именно *сложност* на алгоритъма. На първо приближение ще кажем, че *сложността на алгоритъма* е термин, с който се означава колко са големи "*усилията*" които алгоритъмът прави, за да реши задачата. Усилията се "мерят" съотнесени към големината на "обработваното" от алгоритъма, т.е., в зависимост от големината на входа. В случая усилията на алгоритъма да намери максималния елемент на един масив ще зависят от това колко е голям масивът.

Нека размерът на входа е например n. Този алгоритъм "работи" чрез сравняване стойности.

Както може да се проследи на схемата, ако размерът на входа е n, алгоритъмът ще направи n–1 сравнения. За да се намери Max на масив от 183656214 елемента, на този алгоритъм са му необходими 183656213 сравнявания. По-общо казано, броят действия, необходими за решаване на задачата е пропорционален на броя на елементите. Това е и оценката за сложност на разгледания алгоритъм.

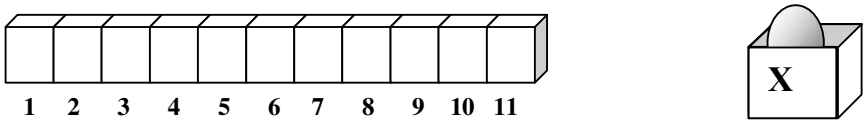
♦ *Примерни задачи – претърсване на масив и умножение на матрици*

➢ *Претърсване на масив*

Друга много често срещана задача е тази за претърсване на структура с данни. Нека в един масив са въведени или са се получили по някакъв друг начин стойности за елементите. Да се претърси масивът означава да се провери дали някаква зададена стойност X е записана някъде в масива. Съществуват множество варианти на тази задача – например: а) има ли такава стойност въобще; б) къде точно, кои са елементите, които имат такава стойност и колко са те; в) кой е първият срещнат при претърсването елемент с такава стойност и т. н.

Един от подходите за решаване на задачата за претърсване е обхождането. Това изисква толкова сравнения на стойностите на елементите на масива със стойността X, колкото е броят на самите елементи. Обхождането е един трудоемък, но сигурен начин за намиране на X, ако го има.

Да се спрем на такъв вариант на задачата за претърсване:
Даден е едномерен масив A.



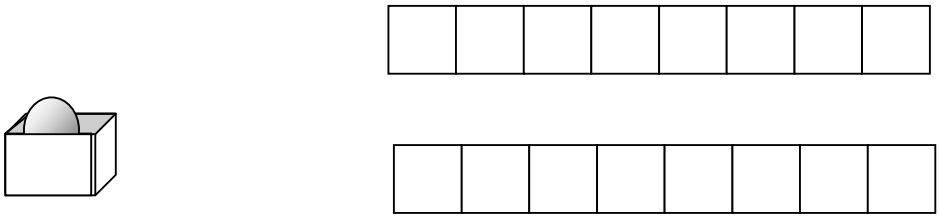
Трябва да се установи дали стойността на променливата X (от същия тип както елементите на A) е стойност и на някои елементи на масива и на *кои точно***.**

Идея за решение.

Нека съставим вектора на принадлежност Pri (нарича се още характеристичен вектор), който да служи като "индикатор" за това на кои места от вектора A се среща стойността на X. Този вектор е също масив, който има толкова места (елемента), колкото са тези на A. Такъв масив често се нарича *паралелен* на изходния. Стойностите на елементите на вектора на принадлежност Pri са както следва:

0 (или false) ако на това място (индекс) от вектора A стойността ***не е X***.

1 (или true) ако на това място от вектора A стойността ***е X.***



Използването на вектора на принадлежност позволява претърсването на А за стойности, равни на Х да се извърши чрез обхождане със следното действие, изпълнявано за всеки "посетен" елемент: Ако текущият елемент A[i] = X, съответният му Pri [i] = 1, в противен случай Pri [i] = 0.

- ***Алгоритъм за умножение на матрици***

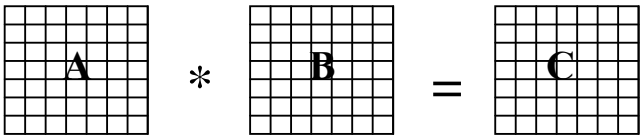
В алгоритмите много често се налага да се извършват операции с матрици. Обичайните операции като умножение на матрици, намиране на обратна матрица, намиране на детерминанта и т.н. са подзадачи, използвани в широк клас алгоритми за разнообразни по произход задачи. Много са задачите, решавани с апарата на линейната алгебра и аналитичната геометрия, например съществена част от алгоритмите, използвани в компютърната графика. Представяне с матрици възниква често в числените методи, в алгоритмите над графи, при прилагането на някои алгоритмични стратегии. Поради това в ще се спрем на един пример – алгоритъма за умножение на матрици.

Ще отбелязваме операцията "умножение на матрици" с " * ".

Умножението на матрици *не е комутативна операция*.

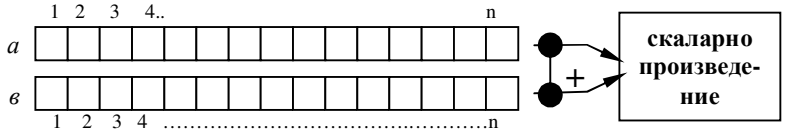
A*B ≠ B*A , където A и B са матрици.

Резултатът от умножението на две матрици A и B *е матрицата* C.



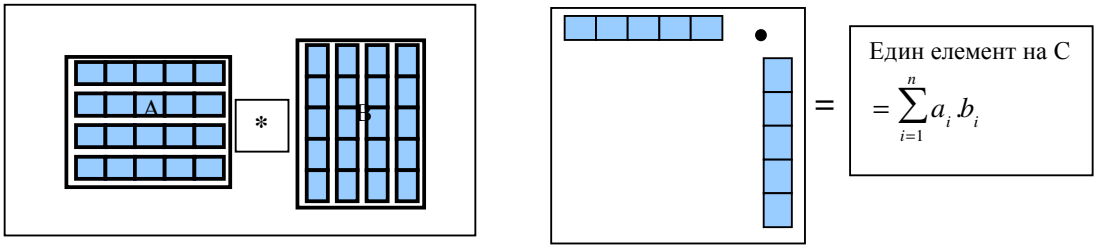
Всеки от елементите матрицата C е равен на *скаларното произведение* на два вектора – един вектор – ред от матрицата A и един вектор – стълб от матрицата B.

Скаларното произведение на вектори е число, равно на сумата от произведенията на елементите с еднакви позиции в двата вектора. Долу е дадена принципна схема на пресмятането на скаларно произведение на два вектора – ***а*** и ***в***. Умножаваните вектори трябва да имат еднакъв брой елементи.



$$a.b = S_{ab} = a_1.b_1 + a_2.b_2 + a_3.b_3 + ... + a_n.b_n = \sum_{i=1}^{n} a_i.b_i$$

За целите на алгоритъма за умножение на матрици, можем да си представим матриците както е илюстрирано долу: Първата, тази отляво на операцията * – като наредени един под друг вектори – редове; втората, тази отдясно на операцията * – като "залепени" един до друг вектори-стълбове.



Ясно е, че не могат да бъдат умножени кои да е матрици. Скаларното произведение на ред по стълб налага броят на елементите в реда на първата матрица да е равен на броя на елементите в стълба на втората матрица. По отношение на другия размер на матриците няма никакви ограничения. Нека пресметнем скаларното произведение $c = \sum_{k=1}^{n} a_k . b_k$

Това става с натрупване на сума в цикъл – за сумата се отрежда една променлива, в която да се добавя всеки следващ член на сумата. Проверете дали даденият вдясно програмен текст прави исканото сумиране.

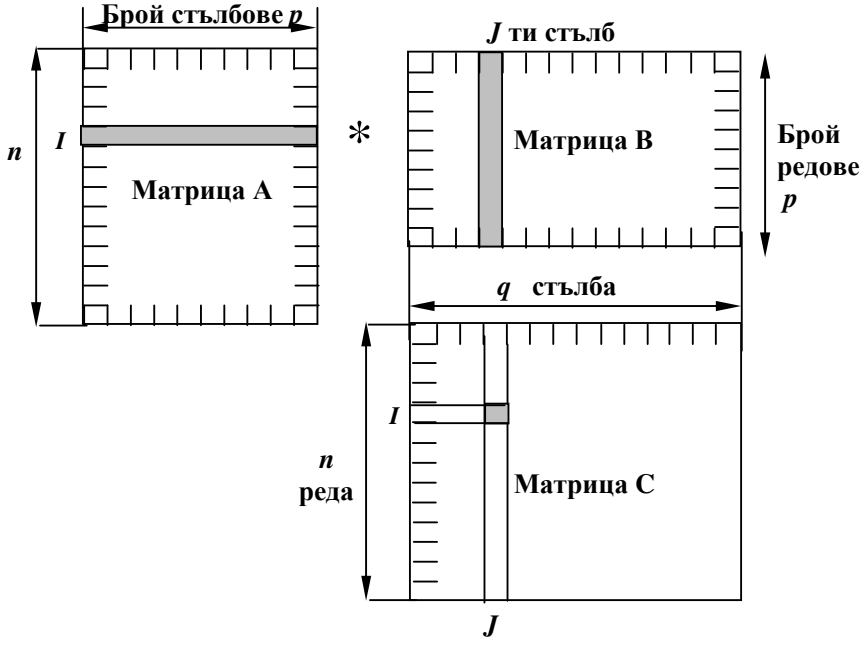
```
C =0;
For k:=1 to n do
  C:=C+a[k]*b[k];
```

```
C=0;
For  (k =0; k<n; k++)
   C+= a[k]*b[k];
```

След като припомнихме как да се пресмята стойността на един елемент от резултатната матрица C, нека да определим и "местонахождението" на всяка натрупана по описания начин сума.

Нека резултатната матрица C е индексирана така: I е индексът, показващ номера на реда, J е индексът, показващ номера на стълба. Тогава всеки елемент C_{IJ} от C е скаларно произведение от I-тия ред на A и J-тия стълб на B.

C_{IJ} = $I_{ти}$ ред от A * $J_{ти}$ стълб от B



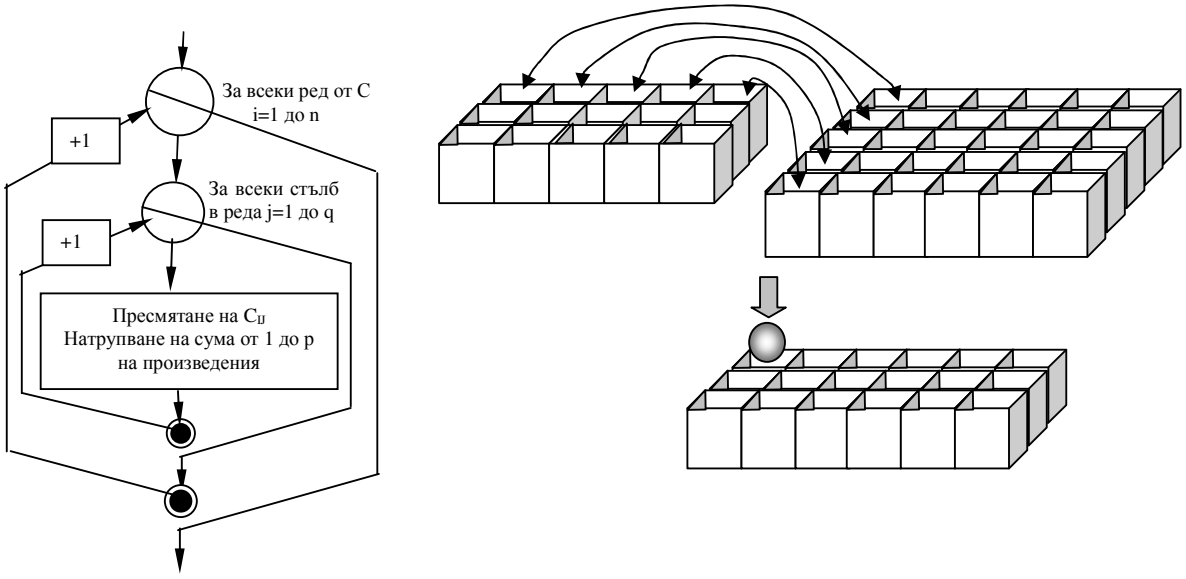
Ако матрицата A е с размери n реда на p стълба и B е с размери p реда на q стълба, резултатната матрица C ще бъде с размери n реда на q стълба.

Алгоритъмът трябва да пресметне един по един всички елементи на матрицата C. Това става с обхождане. Дали да е по редовете или по стълбовете на матрицата C, няма значение. Двата индекса на един елементите на C фиксират един ред на A и един стълб на B.

Скаларното произведение, което се изчислява за всеки елемент C[IJ] е сума, която има p члена. За сумиране постъпват елементи на A и на B, като: елементите на A имат първи индекс I – този на реда, който умножаваме, а тези на B имат втори индекс J – този на стълба, който умножаваме. Общата формула е дадена долу.

$$C_{ij} = \sum_{k=1}^{p} a_{ik} . b_{kj}$$

В тази сума, k е брояч на елементите на двата умножени вектора, които имат по p елемента.



Съставете алгоритъма и процедурата за умножение на матрици.

1.2 Сортиране

- ***Обща постановка***

Задачата за сортиране на данни възниква много често в практиката и това е породило голям брой алгоритми за нейното решаване. Алгоритми, които работят по различни начини и които полагат различни "усилия". Разглеждането на такива алгоритми в учебен курс има голямо методическо значение, поради което тук ще се спрем подробно на множество алгоритми за сортиране, ще навлезем в детайли, ще прилагаме разглежданите вече похвати за изграждане на алгоритъм. Полезно е и въвеждането на някои начални понятия от анализа на алгоритми по сложност.

Най-общо, задачата за сортиране може да се дефинира така:

Еднотипните данни са организирани като стойности на елементите на някаква структура. В множеството на данните съществува релация на наредба. Да се разместят данните така, че разположението им в структурата да съответства на наредбата.

Съществена част от похватите и тънкостите на сортирането се проявява при алгоритмите за сортиране на масив. Затова ще се занимаем подробно със сортиране на масив.

Най-простата структура от еднотипни данни е едномерният масив (вектор). Един вече сортиран във възходящ ред масив например от цели числа изглежда както е показано долу. Това е изходът на задачата.

4	6	7	18	23	28	31	44
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**

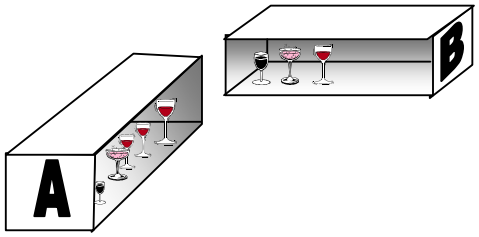
Ако преди да придобият показания нареден вид данните са били стойности на масив, но "разбъркан", то задачата се нарича "сортиране на масив" и фигурира в почти всички учебници по алгоритми и програмиране

Широката популярност и известността на наименованията на повечето методи за сортиране на масив довеждат до неволно пренебрегване на тяхното методическо значение от гледна точка на алгоритмите въобще. Това е причината, поради която в това учебно помагало ще се отнесем с особено внимание към алгоритмите за сортиране на масив, като ще приемаме, че те са достатъчно добре разгледани само след като е изяснено значението на всеки алгоритмичен блок и на всеки оператор от програмния текст.

1.2.1 Сортиране в друг масив

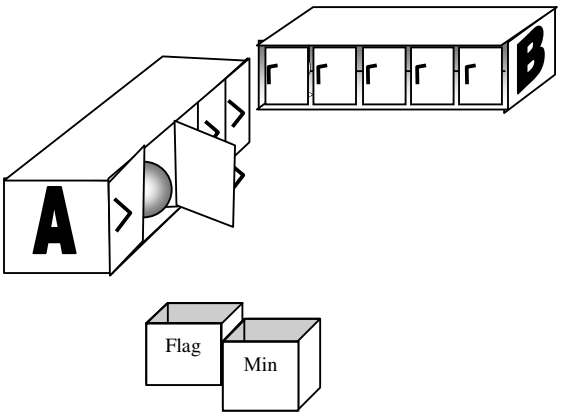
Най-напред ще разгледаме един много непретенциозен алгоритъм за сортиране, който има значение единствено за доброто разбиране на задачата. Нека предположим, че данните (стойностите) от един масив A, който е "разбъркан" трябва да се прехвърлят в *друг масив B,* където да се разположат наредени по големина

Такава постановка е лесна за разбиране и облекчава началната идея за алгоритъма. Нека си представим, както е илюстрирано встрани с чаши, че чашите трябва да се прехвърлят от един рафт на друг рафт, като се наредят по височина. Взема се най-ниската чаша и се поставя на първото място, после – от останалите чаши се взема отново най-ниската и се поставя на второто място и т.н.



Дали идеята за алгоритъм, породена от аналогията с чашите е буквално приложима при масив? Да, принципът е същият. С тази разлика, че както е показано на илюстрацията долу, за масив е малко по-сложно и "по-уморително". Трябва да се обходи масивът A и да се намери минималният му елемент. После намерената стойност трябва да се запише на първо място в масива B.

Трябва втори път да се обходи A, при това без да се взема под внимание "първия намерен минимален елемент", да се намери минималният от останалите и да се постави на второ място в масива B. Трябва трети път да се обходи A, да се намери минималният от останалите... Само за последния елемент не е необходимо да се обхожда, той е единственият "останал непренесен" от A и трябва само да се пренесе в B.



В допълнение на тази образно казано "слепота" на алгоритъма, елементите на масива A, които са вече "пренесени" на полагаемото им се място в масива B, не "изчезват" от местата си в A. Алгоритъмът само е копирал стойностите им в масива B. Необходимо е да се направи така, че при всяко следващото търсене на минимална стойност в A алгоритъмът да "не забелязва" класираните вече в масива B стойности.

Идеята този проблем да се реши, като "класираните" вече елементи на масива A се отбелязват с маркер, не е много удачна. Маркерите трябва да се организират в допълнителен масив. До този извод може да се достигне и при по-внимателен анализ на дадения в предишната тема пример с претърсване на масив, в който се използваше характеристичен вектор, за да бъдат "отбелязани" дадени елементи от масива.

Всъщност, тук целта е при *следващото търсене на минимален елемент на A*, алгоритъмът да не "взема за минимални" стойностите, извлечени при предход-

ните обхождания. Всяка от въпросните стойности е вече съхранена в масива B. Следователно, в масива A тя може да бъде заменена с друга стойност, не влияеща върху извличането на следващия минимален елемент. Коя стойност няма да бъде приета от алгоритъма за минимална? Максималната стойност на елемент от A. Тази малка "хватка" решава проблема – след всяко записване на поредната намерена минимална стойност, елементът на A се "деактивира" като му се присвоява максимална стойност.

```
Program AtoB;
Const Sz=100;
Type Massiv = Array [1..Sz] of integer;
Var A,B: Massiv;
I,N,J,MAX,MIN,Flag : Integer;
Begin
      Readln (N);
      For I:=1 to N do
      Readln (A[I]);
      MAX :=A[1];
      For  I:=2 to N do
      If A[I]>MAX then
          MAX:=A[1];
      B[n]:=MAX;
      For I:=1 to n–1 do
        Begin
        MIN:=A[1]; Flag:=1;
        For J:=2 to n do
          If A[J]<MIN then
          Begin
          MIN:=A[J];
          Flag:=J
          End;
        B[I]:=MIN;
        A[Flag]:=MAX
      End;
   For I:= 1 to N do
   Writeln (B[I]);
End.
```

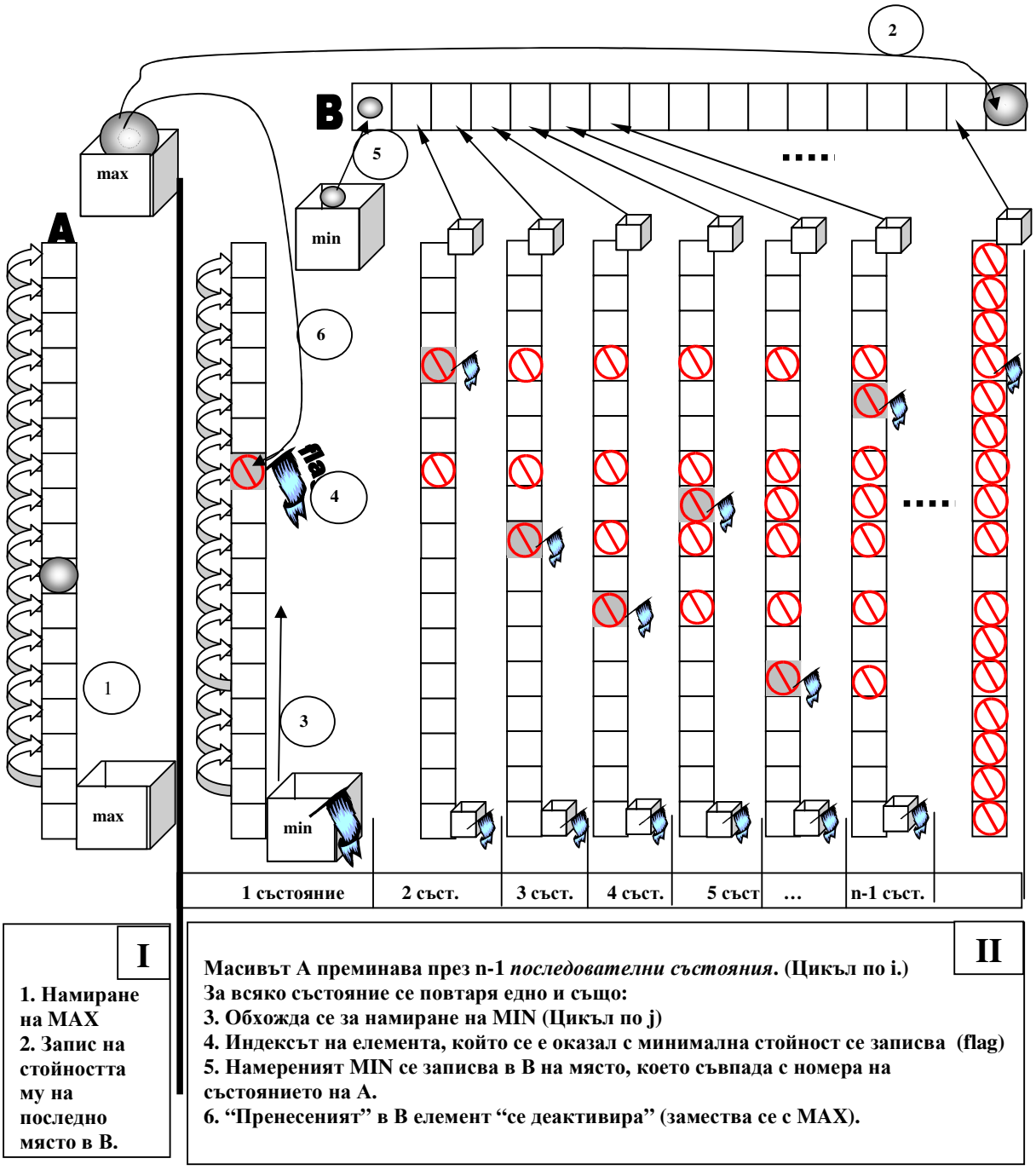
```
int main ()


{int A[100], B[100], I, N, J, Flag, MIN,MAX;

cin >> N;
for (I=0; I < N; I++)
cin >> A[I];
MAX=A[0];
for (I=1; I<N; I++)
if (A[I]>MAX)
  MAX = A[I];
B[N]=MAX;
for (I=0; I<N–1; I++)
{
MIN=A[0]; Flag=0;
for (J=1; J<N;J++)
 if (A[J]<MIN)
 {
 MIN=A[J];
 Flag=J;
 }
B[I]=MIN;
A[Flag]=MAX;
}
for (I=0; I<N; I++)
cout << B[I]);
return 0;
}
```

Горе е даден примерен текст на програмата, реализираща описания алгоритъм. Разчитането на програмния текст не е проста задача дори след като алгоритъмът е описан в общи линии.

Целта тук е да стане ясно какво точно извършва например операторът A[Flag] ← MAX; и да може да се съобрази какво би станало например, ако операторът MIN ← A[1]; се напише три реда по-нагоре.

Ще съставим алгоритъма отначало докрай, като си служим с така наречената *илюстрация на алгоритмичните преобразования*.

Илюстрация на алгоритмичните преобразования над масивите A и B



Масивите A и B имат по n елемента и следователно n пъти в масива B се записват стойности от масива A.

Ще коментираме действията над елементите на масивите A и B, посочени на илюстрацията на алгоритмичните преобразования.

Блок I от илюстрацията на алгоритмичните преобразования:

1. Максималната стойност от A се намира най-напред.
2. Нейното място в масива B не може да е друго, освен последното и затова намерената стойност се записва на мястото си в B.

Блок II от илюстрацията на алгоритмичните преобразования:

Копират се n–1 пъти стойности от A в B. Всеки път "взетата" от A стойност е "поредната минимална" стойност на елемент от A. За всяко записване на стойност в B трябва:

3. да се извлича минимална стойност от A (обхождане),
4. да се пази индексът на елемента на A, който е с минимална стойност,
5. да се записва минималната намерена стойност на съответното поредно място в B,
6. да се "деактивира" пренесеният вече елемент, като му се присвоява максималната стойност на елемент от A.

Масивът A преминава през n–1 *последователни състояния*. В последното си състояние, всички позиции на масива A съдържат стойността на максималния му елемент, т.е. масивът A е "развален" от гледна точка на съхранението на данни.

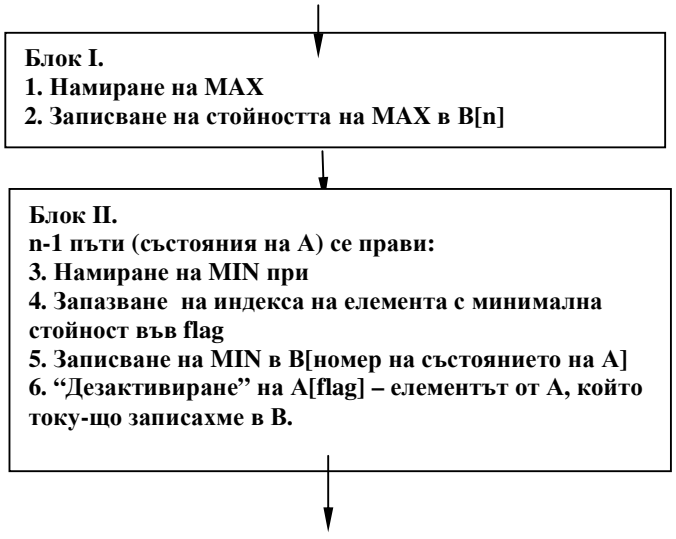
- ***Алгоритъм и програма, съставени по Top-down метод***

Да опишем все пак ниво "нула", т.е. алгоритмичната задача.

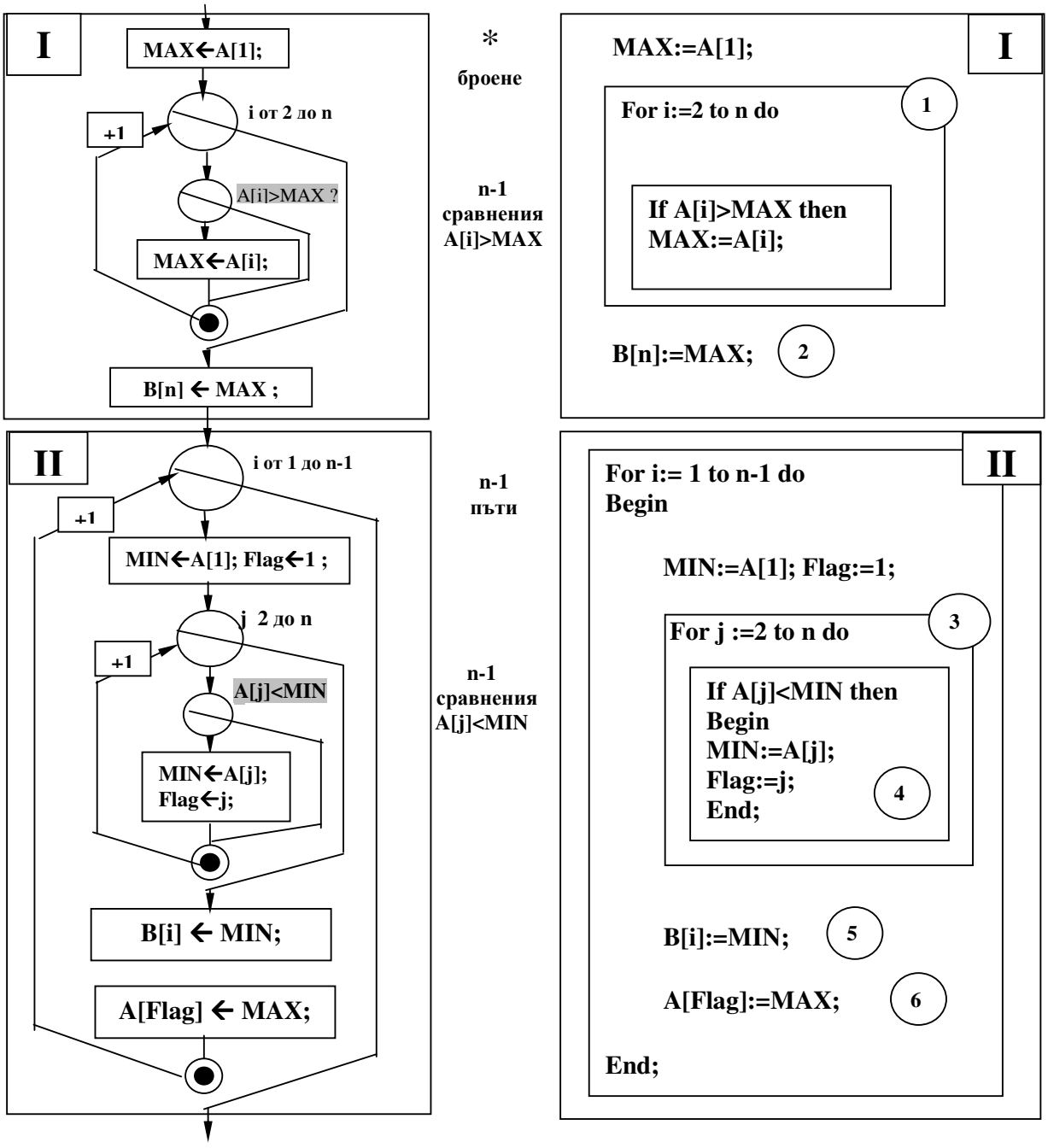
Илюстрацията на алгоритмичните преобразования от предходната страница е основната опора при съставяне на първото ниво на детайлизация на алгоритъма. При детайлизирането се използват същите означения, включително същите номера на алгоритмични действия, които са означени на илюстрацията на преобразованията.

Долу е показано първото ниво на детайлизация. Както се вижда, на български език е описана схемата на преобразованията.

Този алгоритъм съдържа подзадача, за която имаме решение – това е намирането на минимален или максимален елемент на масив. Следователно, съответните блокове трябва просто да се заместят в схемата на управление, като при това се внимава с тяхното взаимно обвързване. В конкретния случай трябва да се съобразят индексите в управляващите цикли.



На дадената долу пълна схема на управление, по индекс i се менят последователните състояния на масива A, а индекс j се мени при обхождането за намиране на минималния елемент в рамките на даденото състояние. Примерният програмен текст на Pascal съвпада с този, който беше представен в началото, преди да се разгледа илюстрацията на алгоритмичните преобразования. Проследете схемата на управление паралелно с илюстрацията на алгоритмичните преобразования:



След направения анализ и виждащото се от схемата съответствие на алгоритмичните преобразования и програмния текст, би могло да се отговори например на следните въпроси:

1. Какъв е алгоритмичният смисъл на оператора Flag:=j?
2. Защо операторът A[flag]:=MAX; е вътре в цикъла на последователните състояния II.
3. Какво би станало, ако операторът MIN:=A[1]; се извади извън цикъла II?
4. Какво би станало, ако операторите с номера 5 и 6 си сменят местата?

Ако програмистът е в състояние да отговори на такива въпроси, той несъмнено може да променя, подобрява и т.н. програмния текст.

- ***Елементи на анализ на сложността на алгоритъма***

Алгоритмите могат да бъдат реализирани на различни машини и при използване на различни езици за програмиране. Въпреки това, един и същи алгоритъм, реализиран на различни езици и дори на принципно различаващи се "абстрактни машини", има собствени, присъщи на самия него характеристики. Една от съществените характеристики на алгоритъма е така наречената "сложност на алгоритъма по време", която в предходната точка оприличихме най-общо на "усилията" на алгоритъма да доведе задачата до решението й, т.е. да преобразува входа в изход.

Сложността по време се изразява най-общо със зависимост между броя стъпки, които алгоритъмът извършва до получаване на изхода на дадената задача, и големината *n* на постъпващия за обработка от този алгоритъм вход.

Както вече споменахме, задачата за сортиране на данни възниква много често в практиката. Когато се прави избор на алгоритъм за сортиране измежду съществуващите, вземат се под внимание редица техни характеристики, основна от които е сложността по време. Когато се съставя нов алгоритъм, това се прави със стремеж той да има добри характеристики, като отново основна е сложността на алгоритъма по време. Поради това, ще се запознаем на съвсем начално равнище с начина за оценяване на алгоритъм. Ще проследим как принципно се оценява сложността по време на базата на разгледания примерен алгоритъм за сортиране.

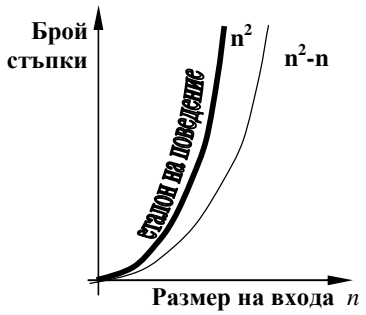
В разгледания алгоритъм масивът A е с определен брой запълнени места – n. Това е размерът на входа – броят на данните, които ще постъпят за подреждане. Очевидно, усилията на алгоритъма зависят от този брой. Както казахме, "усилията" се измерват с *брой стъпки*, които алгоритъмът извършва, за да си "свърши работата". Обикновено, за да се направи тази оценка, се взема под внимание *най-тежката* за изпълнение стъпка.

Разгледаният алгоритъм работи чрез многократно извличане на минимална стойност. Самото намиране на минимална стойност става на базата на многократно сравняване на стойности. Най-общо, алгоритъмът работи чрез *сравнения.* Сравняването на стойности е и най-тежката характерна стъпка, която алгоритъмът извършва. Затова, за приблизителна оценка на неговата сложност по време ще преброим сравненията, които той би извършил, ако на входа са постъпили ***n*** данни. Много удобно е броенето на стъпките да се извършва върху схемата за управление. Погледнете схемата на управление, там сравненията на стойности са отбелязани и преброени. Общо, те са:

$(n-1)+(n-1)*(n-1)$

След като се разкрият скобите, получава се:

$(n-1)+(n-1)^2 = n-1+n^2-2n+1= n^2 - n.$



В горния израз начина на нарастване на броя стъпки в зависимост от нарастването на големината на входа n, се предопределя от члена n^2.: "Поведението" на алгоритъма по отношение на размера на входа ***n***, изразено в брой на извършените стъпки, е *като това на* $\boldsymbol{n^2}$.

Нека приемем, че квадратната функция е нещо като "еталон на поведение" на алгоритъма. Много алгоритми за сортиране се придържат към този "еталон на поведение". Означението $\theta(n^2)$ е означение за сложност по време на алгоритъм, еквивалентна на начина на нарастване на функцията n^2 в някакъв по общ смисъл[1].

Този алгоритъм, който изглежда лош и неефективен, е напълно сравним с много други алгоритми за сортиране, базиращи се на "сравнения при обхождане". Ако има нещо, което го прави наистина лош, това е, че *той разхищава памет*, защото работи с допълнителен масив B. Използваната от даден алгоритъм памет е друга негова важна характеристика и се нарича *сложност по памет***.**

- ***Сортиране без използване на друг масив (сортиране "на място")***

Нека, за да избегнем използването на втори масив, сортираме масива A "в самия него". На входа на задачата масивът A е несортиран, на изхода – същият масив A е вече сортиран. Ще разгледаме множество алгоритми, които са базирани на принципа на многократното *намиране на най-малкия елемент*, но разместват елементите на самия масив A, без да ползват втори масив. За разлика от разгледания преди алгоритъм, преобразованията над един "разбъркан" масив трябва да се организират така, че да се използват по подходящ начин клетките на същия масив. Основната логика на този вид

[1] Означението $\theta(n^2)$ е означение на класа функции, *асимптотично* еквивалентни на n^2

алгоритми е свързана с разместване на местата на стойностите, като при това стойностите очевидно не трябва да се "губят".

Ясно е, че в един сортиран масив мястото на най-малкия елемент трябва да е първо, мястото на втория по големина елемент да е второ и т.н. Следователно, по аналогия с предходния алгоритъм, първото намиране на минимален елемент трябва да завърши с поставянето му на първо място в същия масив, второто – с поставянето му на второ място и т.н. Въпросът е къде да се съхранят стойностите на "досегашния първи", "досегашния втори" и т.н. Очевидно, те се съхраняват някак в масива.

Всяко следващо извличане на минимален елемент води до това, че една стойност "застава на мястото си" в масива. Това последователно добавяне на нови "наредени на местата си " стойности образува една част от масива, която е вече наредена и една, която все още не е наредена. Множество алгоритми за сортиране работят по този принцип, като разликите между тях са в начина на разместване на стойности "към" и "от" наредената вече част на масива.

1.2.2 Сортиране по метода "пряка селекция"

Ето логиката на този алгоритъм, илюстрирана на фигурата долу:

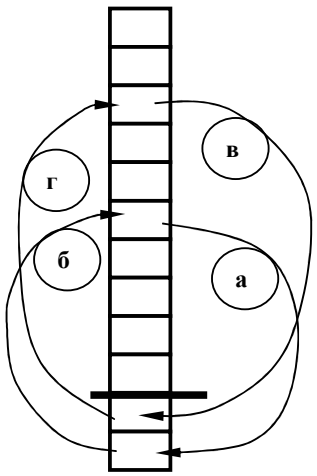
а) Най-малката стойност трябва да се запише на първа позиция в масива, при което позицията, която най-малката е имала, се освобождава.

б) Стойността, която е била записана в първия елемент на масива, трябва да се съхрани, като се запише на мястото, където е била най-малката стойност.

в) "Втората най-малка" трябва да се запише на втора позиция в масива.

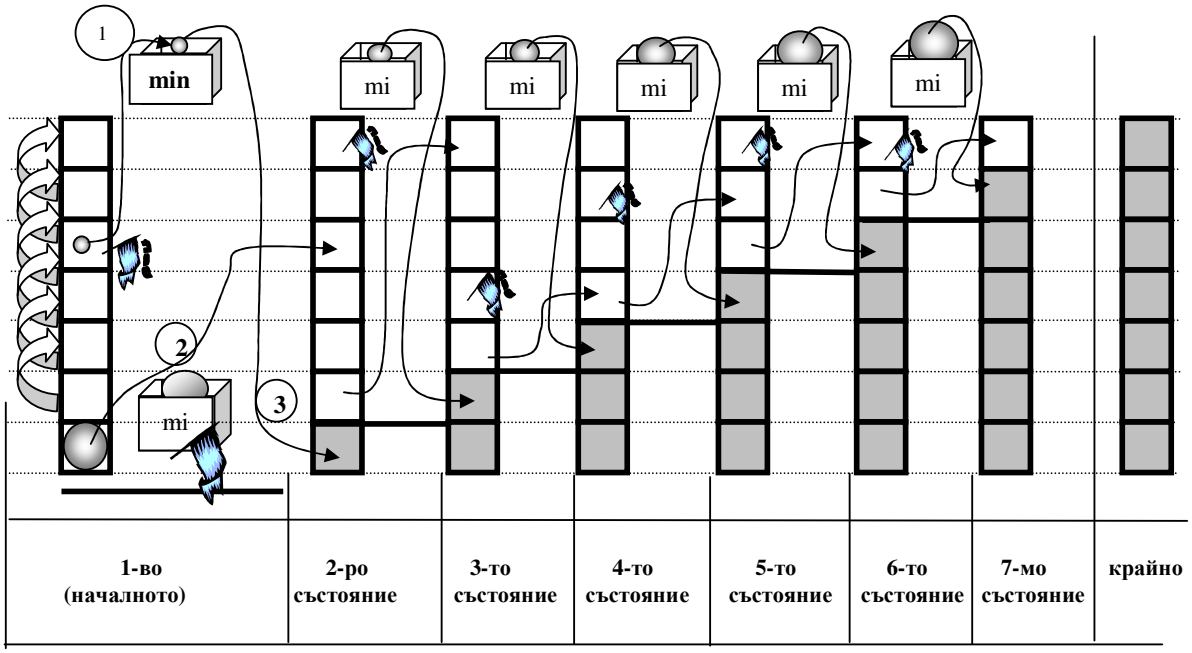
г) Стойността, която е била записана във втора позиция на масива, трябва да се съхрани на мястото на "втората най-малка" стойност.

И така n–1 на брой пъти.



По-нататък е дадена илюстрацията на алгоритмичните преобразования на сортирането по метода на пряката селекция. Обърнете внимание, че алгоритъмът разделя масива A на две части – вече подредена и още неподредена част. В началото неподреден е целият масив, а в края – подреден е целият масив. Щрихованата част на схемата е наредената вече, а останалата част подлежи на обхождане за намиране на минимален елемент.

➢ ***Илюстрация на алгоритмичните преобразования, извършвани от алгоритъма "пряка селекция"***



За да се сортира, масивът преминава през n–1 последователни състояния. Преминаването от едно състояние в друго става след селектиране на минимален елемент от неподредената част на масива и поставянето му на съответното място – последно в подредената част на масива. За целта, във всяко състояние се извършват едни и същи преобразования:

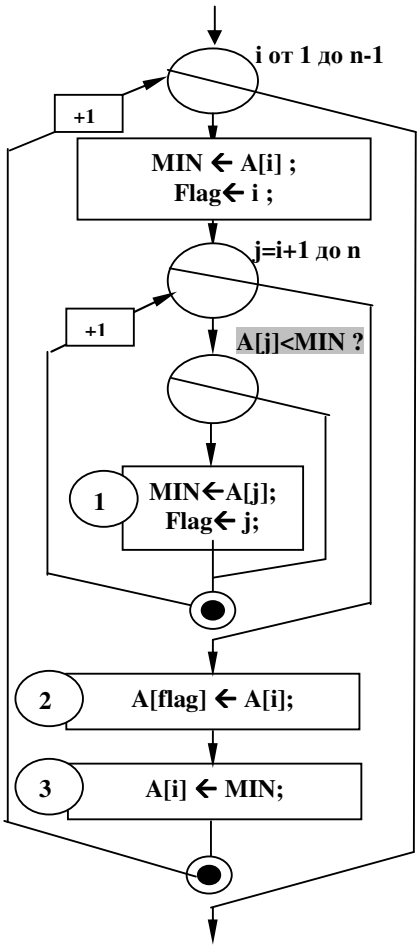
1) Обхожда се неподредената част на масива и се селектира минималният елемент MIN, като мястото му в A се отбелязва с флаг, за да се помни къде е бил той.

2) Освобождава се мястото, на което стойността на MIN трябва да застане. Това място е с пореден номер, равен на номера на поредното състояние на масива. Стойността на елемента, който се намира на това място, *се записва там, където е бил минималният елемент*. Тя остава за по-нататъшна "обработка" при следващите обхождания.

3) Стойността на MIN се записва на полагаемото й се място.

Долу е дадена схемата на управление на този алгоритъм. На нея са отбелязани номера на действията, съответстващи на илюстрацията на алгоритмичните преобразования. Както при предишния алгоритъм, последователните състояния са индексирани с i, а обхождането става с цикъл по j. Ако в този алгоритъм има някаква тънкост, то тя е да се съобрази как обхождането за намиране на минимална стойност да започва всеки път от по-горна позиция. Проследете схемата на управление:

Да съобразим какви са зависимостите между броячите i и j. От схемата на алгоритмичните преобразования се вижда, че за всяко следващо (i-то) състояние обхождането започва от i-та позиция, тъй като предходните i–1 позиции съдържат вече наредени елементи. Това е причината вътрешният цикъл на обхождане (по j) да е от i+1 до n, както и началната стойност на MIN да е A[i].



В тази схема за управление има много малко оператори, чиито места могат да се сменят. Помислете кои са те.

Както може да се прецени от схемата на управление, този алгоритъм също прави от порядъка на n^2 сравнения. Да преброим точно колко пъти се изпълнява тялото на вложения цикъл (сравненията). Това се вижда най-добре на илюстрацията на алгоритмичните преобразования:

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

Поведението и на този алгоритъм се оказва по "еталона" на квадратната функция или още – като това на полином от втора степен.

- ***Сортиране по метода на "потъването"***

Този метод се дава тук с методическа цел, такъв метод не е известен. При него обаче се разграничават ясно логически стъпки, които са в основата на известния "метод на мехурчето", както и някои особености на "чувствителността" на методите за сортиране по отношение на състоянието на входните данни. За да се получи от този метод "мехурчесто сортиране", е необходима само малка модификация.

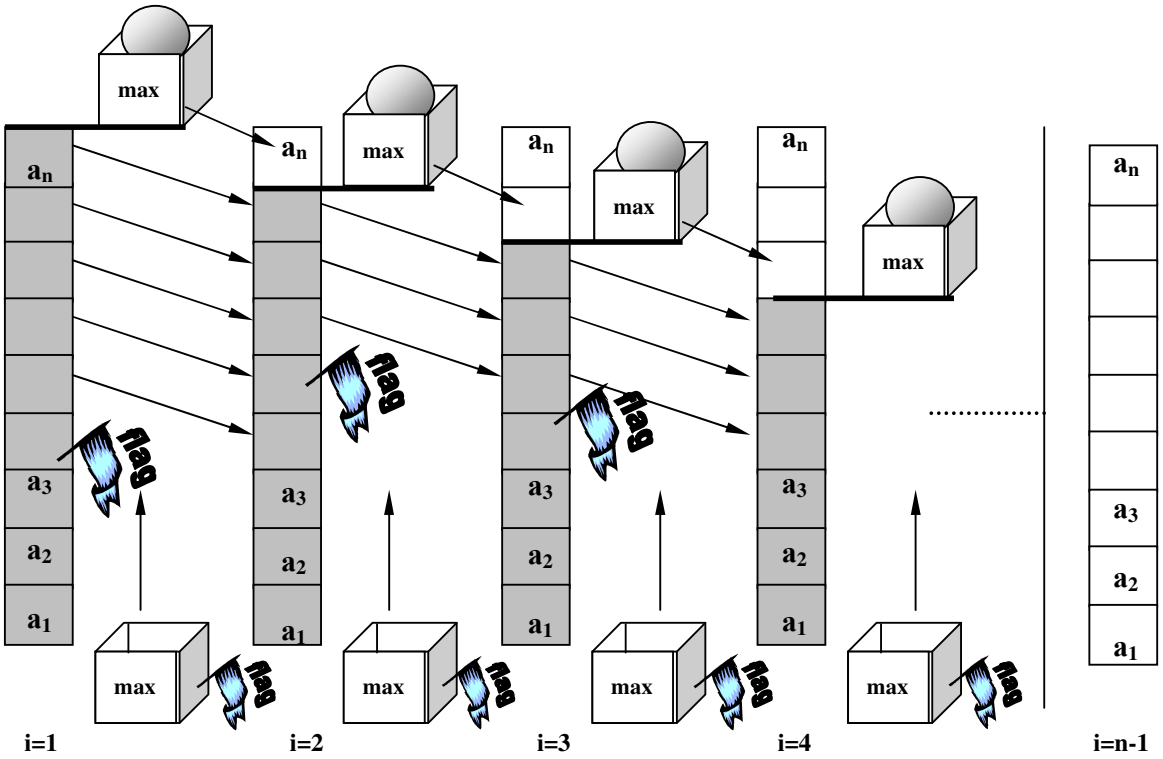
Да разгледаме конкретен вариант на "метода на потъването", който се базира на следната логика: Ако един масив е *вече нареден* и има i елемента, мястото на най-големия е i-то, т.е. последно. Многократното прилагане на това разсъждение, като първо масивът е от n, после от n–1, после от n-2, после ... после от 2 елемента, е метод за получаване на нареден масив от n елемента.

Описание на метода:

Масивът A е логически разделен на две части – една все още неподредена част от n-k елемента и една вече подредена част от k елемента (вж.

илюстрацията на алгоритмичните преобразования долу). Масивът A преминава през n последователни състояния. Първото е даденото начално състояние, в което всичките n елемента образуват "ненаредената част", а n-тото състояние представлява нареден масив A, т.е. целият масив е станал "наредена част". Всяко следващо състояние съдържа "наредена част" над позиции с една повече от броя на подредените позиции в предходното състояние. И, разбира се, обратно – неподредената част "намалява" с по една позиция. От състояние в състояние се преминава n–1 пъти, като всеки път се прилагат едни и същи алгоритмични действия:

➢ ***Илюстрация на алгоритмичните преобразования на сортиране по метода на "потъването"***



Неподредената част на масива (на схемата тя е щрихована) се обхожда и се намира максималната стойност, записана в нея. Мястото, на което тази стойност е била записана, се помни в променливата "флаг".

Всички стойности на елементи от отбелязаното *място на най-голямата стойност* до *края на неподредената част* се местят с една позиция напред, или образно, **"***потъват*" надолу. Това довежда до "освобождаване" на последната позиция на неподредената част. При това "потъване" на мястото, на което е била максималната стойност, се записва стойността на следващия (по-горния) елемент от масива, но това не води до "загуба" на максималната стойност, защото тя е записана в специално предвидената за това променлива MAX.

В последната позиция от неподредената част на масива се записва стойността на MAX.

♦ ***Примерни задачи за упражнение***

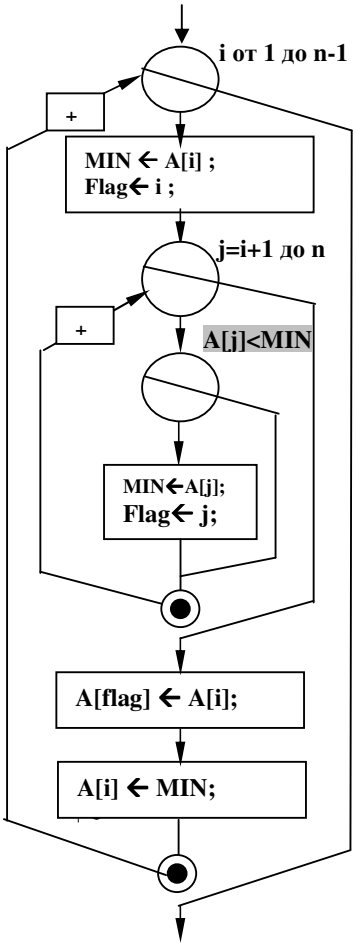
Задача 1.

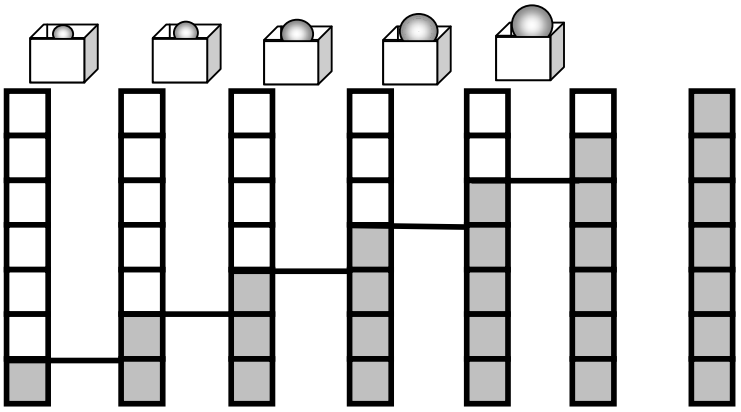
"Проиграване със стойности" на алгоритъма на пряката селекция.

Да се състави илюстрацията на алгоритмичните преобразования, като се предвиди масив с 8 места. Да се *поставят номера* на алгоритмичните блокове и на операторите както на *схемата за управление*, така и на *програмния текст*. Да се попълни масива А с примерни стойности и де се "проиграе" алгоритъмът с тези данни:

А. На илюстрацията на алгоритмичните преобразования: със стрелки, над които е обозначен номерът на предизвикващия преобразованието оператор, да се покаже преместването на стойностите на елементите във второто, третото и т.н. състояния.

Б. Да се трасира работата на алгоритъма в таблица на състоянията на променливите.





1	MIN= …	Flag =…	*обхождане*	A[flag]= …	A[..]= ….
2	MIN= …	Flag =…	*обхождане*	A[flag]= …	A[..]= ….
3	MIN= …	Flag =…	*обхождане*	A[flag]= …	A[..]= ….
4	MIN= …	Flag =…	*обхождане*	A[flag]= …	A[..]= ….
5	MIN= …	Flag =…	*обхождане*	A[flag]= …	A[..]= ….
6	MIN= …	Flag =…	*обхождане*	A[flag]= …	A[..]= ….
7	MIN= …	Flag =…	*обхождане*	A[flag]= …	A[..]= ….

Задача 2

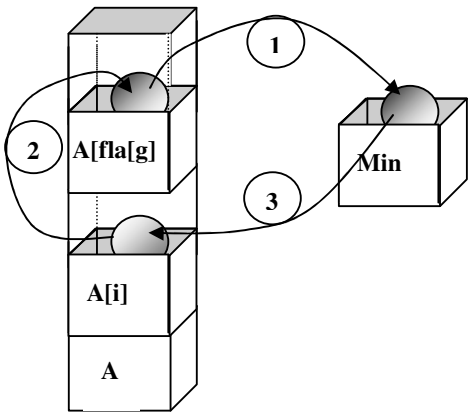
Подгответе сами схемата на управление и програмния текст за алгоритъма на "потъването". Обърнете внимание, че "потъващите"стойности се преместват една по една надолу *в цикъл*, чиято начална стойност на брояча е от мястото, *на което е била най-голямата стойност*, а крайната стойност на брояча е *последната позиция на неподредената част* на масива. Трябва да се изведе

зависимостта, която показва коя е крайната позиция на неподредената част на масива във функция на *n* – броят елементи и *i*-подреденото състояние.

➢ *Чувствителност на алгоритмите за сортиране към наредеността на данните*

Разгледаните дотук методи за сортиране на масив се базират на *сравнения* на стойности и използват подхода на многократно намиране на минимална (максимална) стойност. Два от тях работят с "постъпково" разширяване на наредената част на масива, като това става с разместванията на местата на записаните в масива стойности. Да анализираме смяната на местата. Смяна на местата на две стойности става винаги с помощта на "спомагателна" променлива – "буфер".

На илюстрацията долу е показана последователността на разместванията на стойности в масива A. При смяната на местата на стойностите A[i] и A[flag], променливата min изпълнява функцията на "буферна" променлива.



В контекста на алгоритъма на пряката селекция, смисълът на последователните действия 1, 2 и 3 е:

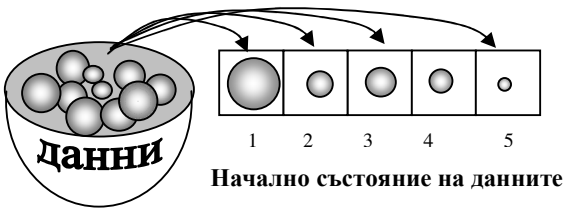
(1) Селекциониране на минимална стойност при запомняне на мястото й в масива.

(2) Освобождаване на позицията, на която селектираната стойност трябва да бъде записана.

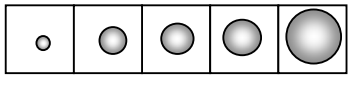
(3) Записване на селектираната стойност на полагаемото й се място.

Смяната на местата се извършва посредством оператори за *присвояване* на стойност. Записването на стойност не струва особени усилия на машината – изпълнител, но е операция, която алгоритмите за сортиране задължително извършват. Ще анализираме връзката между *началното състояние* на данните, попаднали в масива, и броя на *преместванията* на тези данни по различни позиции в масива. Ще установим, че по отношение на този фактор алгоритмите за сортиране "се държат" по различен начин.

Да си представим, както е илюстрирано вдясно, как на входа данните "се разполагат" в местата на масива, в зависимост от *реда на тяхното пристигане.* Целта на сортирането е данните да останат тези, които са въведени, но да са наредени например във възходящ ред.



Начално състояние на данните



Крайно състояние на данните

В крайното състояние на масива между *стойностите* на данните и *местата,* на които те са записани, съществува зависимостта:

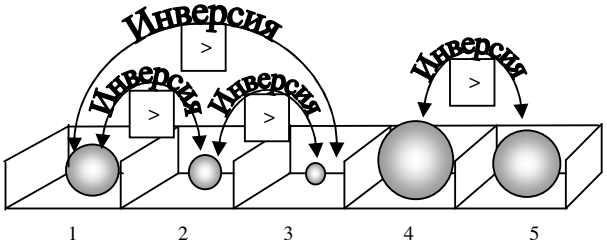
A[i] ≤ A[i+k] (k>0).

Това е условието масивът да е нареден във възходящ ред. В общия случай входният масив е "разбъркан". Както е илюстрирано долу, в него съществуват някои двойки елементи, които не са "правилно наредени".

За тези двойки е в сила:

A[i] **>** A[i+k]) (k>0)

Това нарушаване на търсената наредба в една двойка се нарича *инверсия*. В масива, илюстриран вдясно има четири инвертирани двойки (четири инверсии).

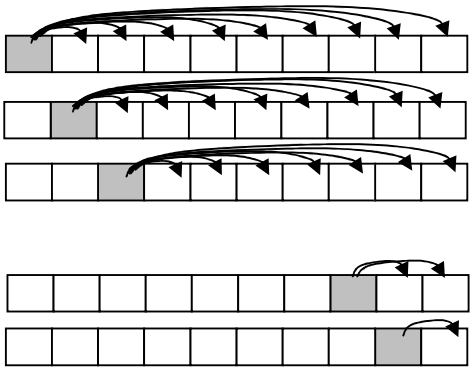


Един алгоритъм за сортиране работи, като в крайна сметка *премахва инверсиите* от началното състояние на данните в масива. Всъщност, задачата на алгоритмите за сортиране е да елиминират инверсии. Логично би било броят на инверсиите във входния масив да оказва влияние върху броя на извършените от алгоритъма размествания. Най-голям брой на инверсии би имал входен масив, който е нареден "наопаки". Да изразим този брой за n входни данни.

Ето една опорна схема за преброяване, която се използва често:

A[1]
е инвертиран спрямо всички останали n–1 елементи
A[2]
е инвертиран спрямо всички останали n-2 елементи
A[3]
е инвертиран спрямо всички останали n-3 елементи
……

A[n–1]
е инвертиран спрямо 1 елемент (A[n])



Общо, инверсиите са: $(n-1)+(n-2)+(n-3)+\ldots+3+2+1=\sum_{i=1}^{n-1} i = \frac{(n-1).n}{2} = \frac{n^2}{2} - \frac{n}{2}$

Това е най-големият брой инверсии, които един масив от n елемента би могъл да има. Да разсъждаваме по принцип: всяка една от инверсиите може да се елиминира, като елементите от инвертираната двойка си разменят местата. Това означава, че за да елиминира инверсиите в един нареден наопаки входен масив, алгоритъмът ще направи от порядъка на n^2 размествания.

Броят на инверсиите във входния масив е "неизбежно зло", защото във всички случаи алгоритъмът ще *трябва да преодолее тези инверсии***.** Ако има инверсии, няма как, те трябва да се премахнат. Би било добре обаче алгоритъмът да "е чувствителен към състоянието на входния масив" и ако *няма инверсии,* да не извършва напразни действия, а да "си почива".

Да анализираме от тази гледна точка разгледаните три метода.

(1) *Сортиране в друг масив.* Този алгоритъм не компенсира инверсии, защото мести в друг масив. Той е напълно нечувствителен към броя на инверсиите и дори да му се подаде сортиран входен масив, алгоритъмът безропотно ще извърши всичките си обхождания и сравнения и ще записва в изходния масив точно n пъти.

(2) *Сортиране по пряка селекция.* Този алгоритъм работи така: n–1 пъти поставя на "правилното" място една стойност, като разменя местата на две стойности. Това е съвсем просто, не изисква кой-знае какви усилия и е убедителен аргумент в полза на този алгоритъм. Едно поставяне на "правилното" място води до *премахването на всички инверсии, които поставяната на мястото си стойност е имала* (в подредената част на масива няма инверсии по дефиниция, а поставената "на мястото си" стойност е по-малка от всички стойности на елементи на неподредената част и следователно не участва в нито една инвертирана двойка).

За съжаление този алгоритъм "не реагира" на състоянието на входния масив. Независимо от това дали в началното се състояние масивът е бил "наопаки", или не е имал инверсии, алгоритъмът ще извършва точно едно и също. Ако масивът е бил нареден, алгоритъмът пак ще го обхожда, ще търси MIN … и ще го връща обратно там от където го е взел. И така – n–1 пъти.

(3) *Метод на потъването.* Този метод се разглежда тук като пример за съчетание на недостатъци, които един алгоритъм за сортиране може да има. Както при пряката селекция, поставят се "на правилното място" стойностите от масива и при всяко поставяне се компенсират всички инверсии на поставяната стойност. Алгоритъмът, обаче, извършва "безумни" действия (които кръстихме "потъване"), за да освободи въпросното "полагаемо се място". Всъщност най-неефективната страна на това "потъване" е това, че то запазва взаимното разположение на потъващите стойности, заедно с инверсиите.

Все пак има и нещо положително в този алгоритъм – той е чувствителен към инверсиите на "входния масив". Ако входният масив *няма инверсии,* алгоритъмът ще извърши всички действия, но *без потъването*. Проследете илюстрацията на алгоритмичните преобразования на метода на "потъването", където се вижда, че цикълът на потъване има граници от Flag до n-i. В случай, че масивът е нареден, извличаната от неподредената му част "поредна максимална стойност" се намира на (n-i+1) -в място, т.е. Flag = n-i+1. Не се налага освобождаване на мястото за намерената максимална стойност, защото тя си е на мястото и фазата "потъване" от алгоритъма не се изпълнява. Програмно цикълът на потъване няма да се изпълнява, защото долната му граница надвишава горната му граница.

Да разгледаме още един алгоритъм за сортиране, чувствителен към инверсиите във входния масив. Той реализира *метода на пряката размяна* или още "*метода на мехурчето*". Този метод не може да се похвали с нищо друго освен със звучното си име и в повечето учебници той е квалифициран като много лош, дори най-лошият. Това е, защото авторите на учебниците на са съставяли умишлено още по-лоши методи и методът на мехурчето не изпъква с някои свои достойнства. Както споменахме, методът на мехурчето

представлява някакво подобрение на дадения тук с методическа цел “метод на потъването”.

1.2.3 Сортиране по метода на пряката размяна (на мехурчето)

Принципна постановка на метода на пряката размяна:

1) Масивът преминава през последователни състояния.

2) Масивът е разделен на подредена и неподредена част.

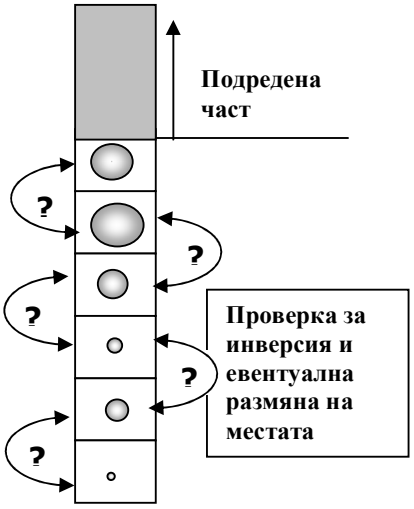
3) Неподредената част се обхожда, като неявно се търси максимален елемент и се поставя на мястото му.

Ще разясним този метод при използване на въведените дотук означения и най-вече на означенията от илюстрацията на метода на потъването.

Както при метода на потъването, този алгоритъм извлича максималната стойност от неподредената част на масива и я поставя на нейното правилно място, т.е. на (n-i+1) -во място.

Той обаче не търси максималната стойност по обичайния начин. Алгоритъмът извършва нещо като “слалом” сред стойностите на неподредената част, като при това “провира” най-голямата стойност нагоре, докато не я постави на правилното й място. За всяко от последователните състояния на масива това преместване на максималната стойност става по следния начин:

От началото на масива до края на неподредената му част, стойностите на елементите се сравняват по двойки (с припокриване): 1 с 2; 2 с 3; 3 с 4; и т.н.т до (n– i)-тата с (n-i+1) -вата.



i-то състояние

Ако при сравняването по двойки се окаже, че в дадена съседска двойка има инверсия, то инверсията се компенсира “начаса”, като двете съседни стойности си разменят местата. Долу е даден примерният програмен текст по блокове, реализиращ обхождането за i-тото състояние на масива. Номерата на състоянията са индексирани с i, като в примерния текст на C, без това да е задължително, за да се следва по естествен начин формализма на езика, индексацията на n-те елемента от масива започва от 0 и завършва с n–1.

```
For j:=1 to n-i do
Begin
  If A[j+1]>A[j] then          | ПРОВЕРКА ЗА ИНВЕРСИЯ И РАЗМЯНА
  Begin
  Buffer:=A[j];
  A[j]:=A[j+1];
  A[j+1]:=Buffer
  End;
End;
```

ОБХОЖДАНЕ НА НЕПОДРЕДЕНАТА ЧАСТ

```
for (j=0; j< n-i ; j++)
{
  if (A[j+1]>A[j])             | ПРОВЕРКА ЗА ИНВЕРСИЯ И РАЗМЯНА
  {
  Buffer=A[j];
  A[j]=A[j+1];
  A[j+1]=Buffer;
  }
}
```

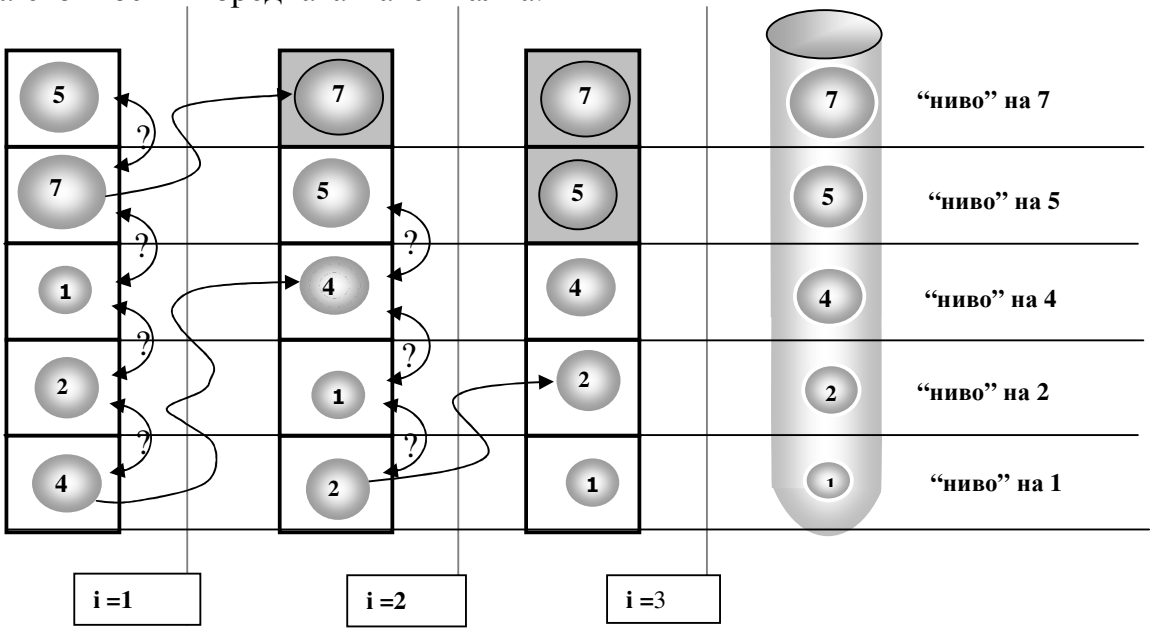
ОБХОЖДАНЕ НА НЕПОДРЕДЕНАТА ЧАСТ

Всяко следващо сравняване на двойка се прави след като предходната двойка е била проверена за инверсия и “разместена” така, че по-голямата стойност да е отгоре. Това довежда до “изплуване” на най-голямата стойност най-отгоре в неподредената част. Така при всяко поредно обхождане на неподредената част, максималната стойност, която е записана в нея, “застава на правилното си място”. Такъв краен ефект се постига и при метода на потъването. Тук, освен че най-голямата стойност “изплува” най-отгоре, “пътем” се компенсират някакви “локални” инверсии чрез смени на местата. Това не е за пренебрегване. За едно обхождане се върши повече полезна работа:

1. Компенсират се всички инверсии на максималната стойност (включително спрямо неподредената част на масива), защото тя се поставя на мястото й.
2. Компенсират се и други инверсии в неподредената част.

Второто може да доведе до “по-бързо” поставяне на всички стойности по техните места.

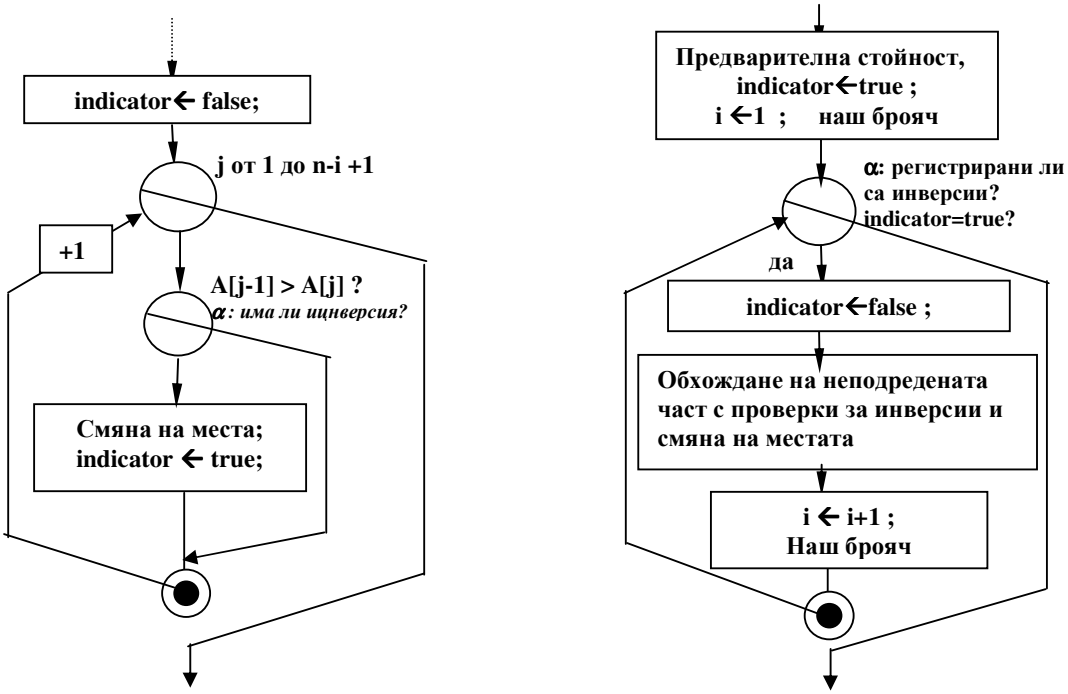
Илюстрация на алгоритмичните преобразования на метода на мехурчето е дадена долу с примерни данни – целите числа 1, 2, 4, 5 и 7. Проследете илюстрацията. Направена е аналогия с епруветка, в която стойностите са оприличени на мехурчета, всяко застанало на своето “правилно ниво”. Забелязва се, че при всяко обхождане по описания начин, “поредната максимална” стойност застава на мястото си. Междувременно, обхождането със смяна на местата довежда до “изплуване” нагоре и на други стойности освен максималната, за сметка на “потъване надолу” на стойности, които също не са били на “своето правилно ниво”. Ако проследите внимателно примера ще забележите, че когато в този процес на разместване някоя от стойностите *застане случайно* на “своето правилно ниво”, алгоритъмът не я мести повече. Следователно, при всяко обхождане с разместване, на мястото си застава *поне една* стойност – поредната максимална.



Вижда се, че не е необходимо масивът да премине непременно през n–1 състояния, за да е сортиран. Когато няма повече инверсии за компенсиране, т.е.

когато всяко мехурче е застанало на "нивото си", не е необходимо да се продължава. Приятната страна на този алгоритъм е, че той може много лесно да "усети", че няма повече инверсии, защото той така "работи" – търси съседски инверсии. Естествено, алгоритъмът трябва да "следи" специално за наличието или липсата на инверсии. Въпросът е дали *въобще има инверсии или няма*, т.е. отговорът е от тип Да/Не.

Да поставим "индикатор" вътре в блока, който се изпълнява в случай, че в двойката е регистрирана инверсия, както е показано на схемата долу вляво. Да въведем например една булева променлива "*indicator*", която да показва "*има ли инверсия*?". Преди началото на всяко обхождане на неподредената част на масива, трябва тази променлива да бъде инициализирана със стойност *false* – "*няма регистрирана инверсия*". В края на обхождането индикаторът ще показва дали е имало инверсии. Ако е имало, текущото i-то състояние на масива трябва да "се обработва" още, за да се получи (i+1) -во състояние. Ясно е, че това може да продължава най-много докато целият масив се превърне в "подредена част", т.е. обхождането може да се наложи най-много n–1 пъти.



Целта на подобрението, въвеждано с индикатора за инверсии, е алгоритъмът да спре веднага, щом се установи че в неподредената част няма повече инверсии. Това би било така, ако променливата *indicator* е останала "*false*" при поредното обхождане на неподредената част. Логично е цикълът, придвижващ масива от състояние в състояние, *да не бъде по вграден брояч i*, а да се трансформира в итеративен цикъл.

На схемата горе вдясно е илюстрирана реализация с *предусловие*. Помислете дали не е по-добре да се състави цикъл със следусловие. Смисълът на условието е: "*Регистрирани ли са инверсии при последното обхождане?"* Да напомним, че всеки итеративен цикъл с предусловие изисква задаване на стойности за участниците в условието α преди да се влезе в цикъла. Каква да

бъде предварителната стойност на индикатора, се преценява така: за да започне да се изпълнява цикълът на последователните състояния (този с брояч i), индикаторът трябва да бъде с предварителна стойност "*има инверсии*".

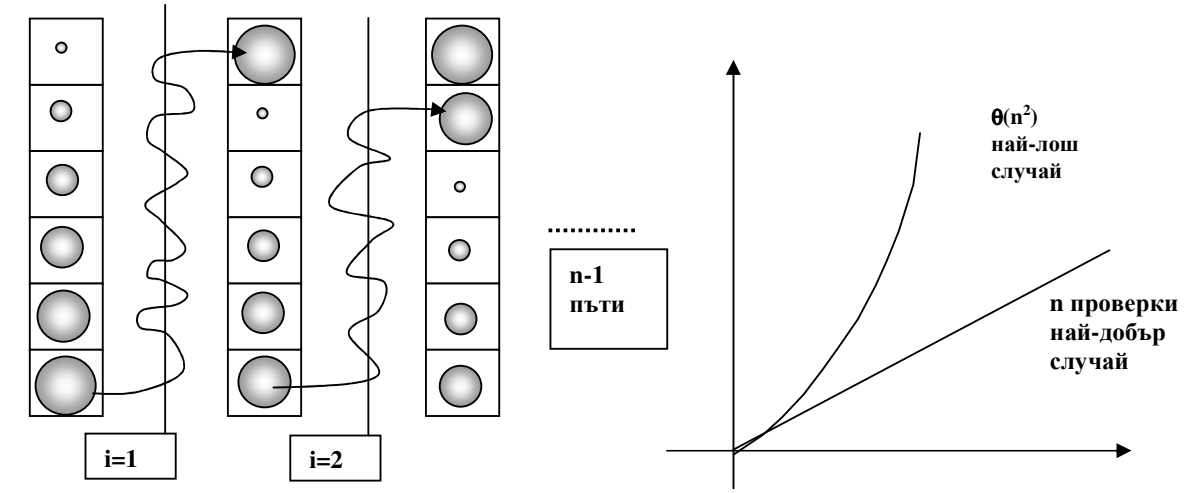
При управление на "мехурче" със спиране по индикатор, се налага нарочното вграждане на брояч i, тъй като итеративният цикъл не поддържа вграден брояч за (поредното състояние на масива). Ако този брояч не се постави, вътрешният цикъл на обхождане (по j) не може да се управлява с граница до (n - i +1), т.е. до края на неподредената част.

Да направим анализ на описания метод за сортиране. Лошите му страни са ясни – за да постави един елемент на мястото му не извършва проста смяна на местата на две стойности както "пряката селекция", а размества много други стойности.

Нека да анализираме и добрите страни на този алгоритъм. По-общо, те са, че той компенсира "съседски" инверсии при всяко обхождане и най-неочаквано може да сортира масив от 100 елемента за два – три хода. Ако входният масив е напълно подреден, алгоритъмът ще направи само n–1 сравнения, ще установи, че няма инверсии и ще спре. Това е неговата сложност *в най-добрия случай* – $\theta(n)$.

За съжаление, в най-лошия случай, т.е. когато масивът пристига за обработка "нареден наопаки", алгоритъмът се държи твърде лошо – като алгоритъма "потъване". Ще "води" нагоре мехурче по мехурче n–1 пъти, като при това всички останали ще потъват с една позиция надолу.

Този алгоритъм може да бъде оценен за два гранични случая на нареденост на входните данни. Той "се държи" много добре в случай , че му е подаден масив без инверсии, и много лошо в случай, че му е подаден обратно нареден масив. На илюстрацията долу е показано поведение на "мехурче" при обратно нареден масив и графика на двете оценки – за най-добрия и най-лошия случай.



Мехурче на обратно нареден масив

Може да се заключи, че методът на мехурчето е подходящ, когато е предварително известно, че подаваните на входа данни са частично наредени.

Тогава неговото поведение ще бъде по-близко до оценката за най-добрия случай, която е наистина много обнадеждаваща.

1.2.4 Сортиране по метода на прякото вмъкване

Методът на прякото вмъкване съчетава добрите страни на разгледаните дотук методи, защото си пести усилията по много "хитър" начин. Освен това, той се поддава на съществени подобрения, за разлика от методите, разгледани дотук. Вместо да обхожда неподредената част на масива *изцяло*, както правят предходните алгоритми, той обхожда *подредената част* на масива. Това му пести усилия по обхождането, защото не винаги се налага подредената част да се обхожда изцяло. Освен това, този алгоритъм е чувствителен към инверсии.

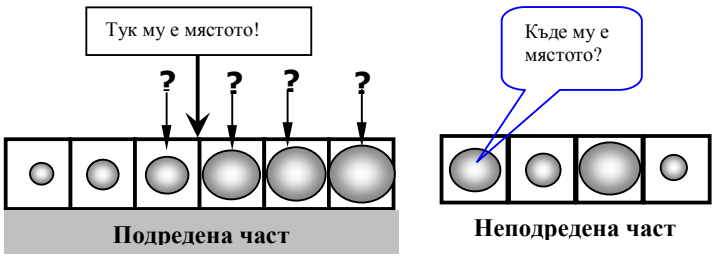
Въпреки че изложената обща постановка на този алгоритъм изглежда някак "обърната наопаки", той е изграден на базата същия скелет, като предходните алгоритми:

1. Масивът от n елемента преминава през n–1 последователни състояния.
2. Във всяко състояние, масивът е "разделен" на две части – подредена и неподредена.
3. За всяко състояние на масива, подредената част се разширява с една позиция, като една стойност от неподредената част *се вмъква* в подредената част.

Основната идея на този алгоритъм е във вмъкването на стойност, затова ще се спрем на логиката вмъкването.

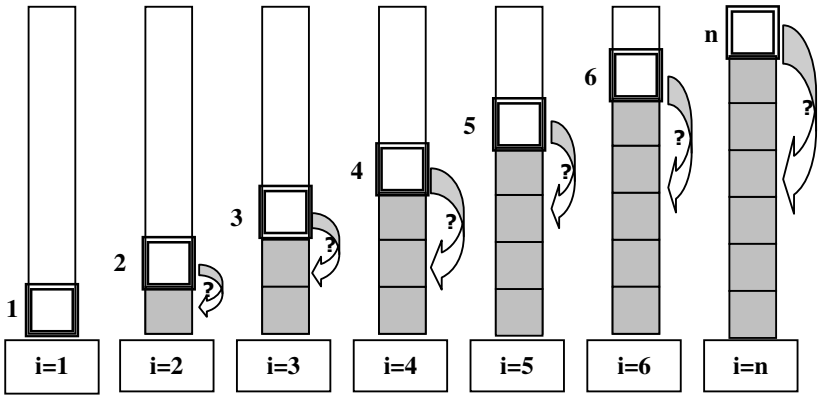
Нека един масив от K елемента е вече запълнен със стойности, наредени във възходящ ред, както е илюстрирано долу. Налага се масивът да "нарасне" с още една стойност, като при това остане нареден. Да си представим, че новопристигащата стойност "си търси мястото" в масива.

Мястото на вмъкваната стойност е преди всички стойности, които са по-големи от нея. Освен това, то е *точно след елемента*, чиято стойност е по-малка или равна на вмъкваната стойност.



На илюстрацията горе са показани двете части на масива, като стойността, която се вмъква, е първата стойност от неподредената част. Стои въпросът как да се освободи "полагаемото се място" за вмъкваната стойност. Както се вижда от илюстрацията, логично е всички по-големи стойности "да се избутат" с една позиция надясно, като най-голямата от тях заема позицията, освободена от вмъкваната стойност. При това те трябва да запазят наредбата си. Това преместване прилича на "потъването", но мести част от масив, в която няма инверсии.

Следващата схема е помощна – това е схема за връзката между поредното състояние на масива и индекса на вмъквания елемент. На схемата, началното състояние на масива е с номер едно. Във второ състояние се вмъква вторият елемент, като "наредената част" се състои от един елемент. В трето – вмъква се третият елемент в частта, в която първите два са вече наредени и т.н. Всяко следващо (i+1) -во състояние се получава от предходното i-то, като стойността на i-тия елемент се вмъква на правилното място в подредената част. Масивът преминава през n–1 състояния, от второ до n-то, като в последното състояние се вмъква последният останал невмъкнат елемент, n-тият.

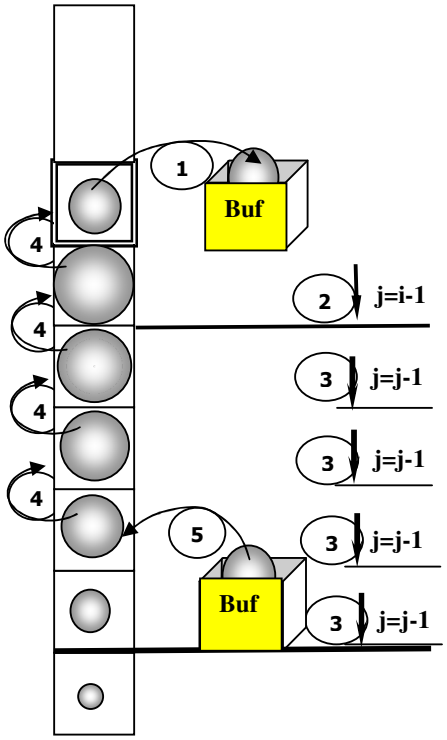


Вмъкването изисква сравнения на стойността, която ще се вмъква със стойностите от подредения под-масив. Долу е илюстриран един начин на вмъкване.

В i-тото състояние на масива, стойността от i-тата позиция се копира в специално предвидена "буферна" променлива. Търсенето на "правилното място" за стойността, намираща се в буферната променлива, става отгоре надолу, стъпка по стъпка. При "слизането" се следи дали стойността на променливата-буфер е по-малка от стойността от масива, намираща се на даденото "стъпало". Ако е по-малка, слизането продължава.

Нека "постъпковото слизане" да има свой брояч j, с който да се сменят индексите на елементите от масива. Стойността "си е намерила мястото" и слизането трябва да спре, когато:

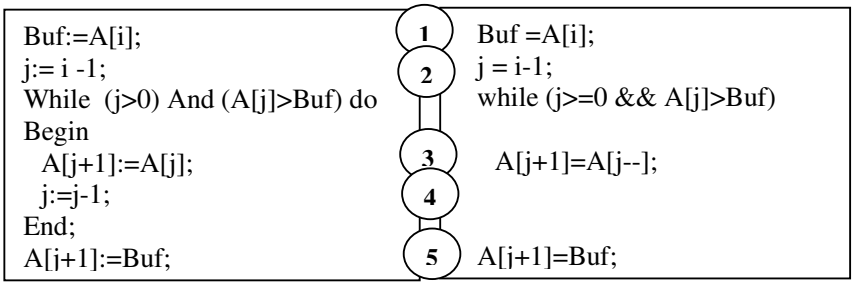
- или елементът от подредения под-масив се окаже (вече) със стойност по-малка или равна на стойността, за която се търси място,
- или когато се стигне до началото на масива (j=1 за Pascal и j=0 за C).



i-то състояние

Щом като текущата j -та позиция не отговаря на това условие за "правилно място" на вмъкване, слизането прави следваща стъпка надолу.

Стойностите от масива се изместват нагоре една по една, паралелно на описания процес на "слизане" надолу за търсене на място на вмъкване. Когато се окаже, че търсеното място не е на текущото j-то стъпало, стойността от това стъпало се измества на по-горната позиция. По-долу е приведен примерен програмен текст на два езика, който съответства на схемата. "Слизането" е реализирано с цикъл с предусловие. Проследете съответствието на операторите в програмния текст и илюстрацията. Да обърнем внимание на факта, че е възможно компилаторът да организира такова изпълнение на And, че при неистинност на първия операнд, останалите операнди да не се изчисляват. Помислете дали именно на това се базира предлаганата долу програмна реализация. Помислете какво би станало, ако в примерния програмен текст двата свързани с And булеви израза на цикъла с предусловие си сменят местата.



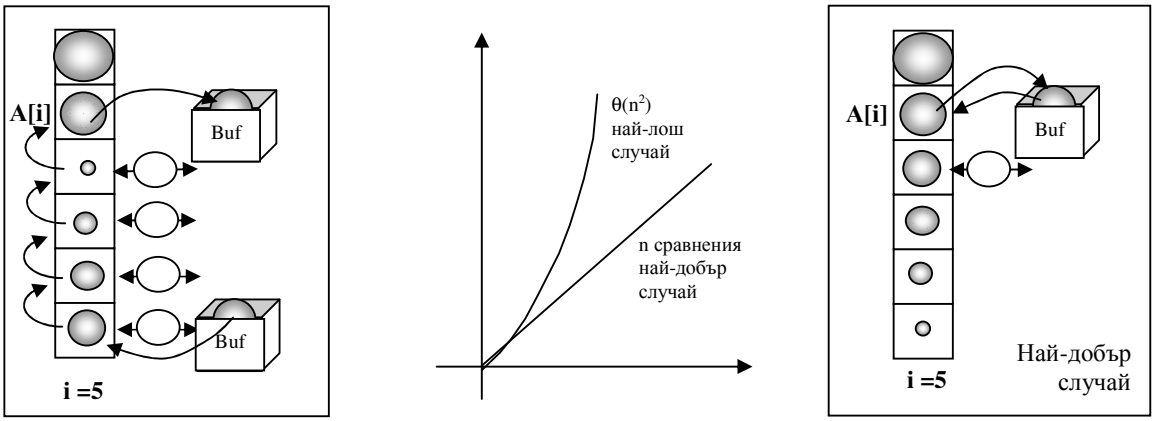
Да направим анализ на метода на прякото вмъкване. Този алгоритъм е чувствителен към инверсии. Броят сравнения, които той прави при "слизането надолу", се оказва пропорционален на броя на инверсиите във входното състояние на масива. Това се вижда и в програмния текст, където условието за край на "слизането" е проверка за това дали вмъкваната i-та стойност на масива е инвертирана спрямо j-тата стойност. Стъпки надолу се извършват толкова пъти, колкото инверсии има вмъкваната стойност спрямо елементите на подредения под-масив. Броят на проверките, извършвани от алгоритъма отново може да се оцени за два гранични случая на нареденост на входния масив.

Най-много са проверките, когато входният масив е нареден "наопаки". Тогава, за да открие "вярното място" на стойността, която вмъква, за всяко i -то състояние алгоритъмът ще прави i–1 сравнения. Мястото на вмъкване ще се оказва винаги първо. Сумарно за всички последователни състояния, алгоритъмът прави $1+2+3+...+(n-2)+(n-1)=\sum_{1}^{n-1} i=\frac{(n-1)n}{2}$ сравнения.

На този брой проверки съответстват точно толкова *премествания* на стойности нагоре. Колкото сравнения – толкова премествания. В най-лошия случай алгоритъмът не само извършва от порядъка на n^2 проверки, но и от порядъка на n^2 премествания. Тогава той се оказва много сходен с алгоритъма на потъването, който мести точно същия брой пъти.

В най-добрия случай масивът пристига за сортиране с 0 инверсии. Тогава алгоритъмът ще "прекара" масива през n–1 последователни състояния, като за всяко състояние ще "установява", че стойността, която трябва да се вмъква, си е

на мястото и ще я "връща" обратно там, откъдето я е "взел". Това изисква една единствена проверка. Ако на алгоритъма се подаде нареден масив, той ще "установи" това след n–1 сравнения и ще спре.



За разлика от другите методи, този алгоритъм обхожда *подредената част* на масива. Това, че се обхожда подредената част има голямо значение за принципната му постановка, защото работата с наредена структура много лесно се поддава на подобрения. В следващата тема ще се запознаем с един начин на претърсване в нареден масив, който може да се приложи за този алгоритъм и подобрява качествено оценката му относно броя на сравненията в най-лошия случай.

♦ *Примерни задачи за упражнение*

Задача 1.

```
Program InsertionSort;
Const Sz = 100
Type vector = array [1..Sz] of integer;
Var G:vector;
T,M,K,As : integer

Begin
    readln (T);
    For M:=1 to T do Readln (G[M]);
    For M:=2 to T do
        Begin
        As :=G[M]; K:=M–1;
       While (K>0) And (G[K]>As) do
            Begin
            G[K+1]:=G[K]; K:=K–1;
            End;
        G[K+1]:=As;
        End;
For M:=1 to T do
          Writeln(G[M]);
End.
```

```
Int main();
{
int G[100], t, M, k, As;



    cin>>t;
    for (M=0; M<t; M++) cin >>G[M]);
    for (M=1; M<t; M++)
       {
        As =G[M]; k=M–1;
        while (k>=0) && (G[k]>As)

            G[k+1]=G[k--];

        G[k+1]=As;
       }
for (M=0; M<t; M++)
          cout<<G[M];
return 0;
}
```

В предходната таблица е приведен примерен програмен текст, лишен от украси и подробности от типа на подсещания и защити. Той реализира "скелета" на алгоритъма на прякото вмъкване. Разпознайте алгоритмичния смисъл на всеки оператор и блок в този програмен текст, като спазвате следната последователност:

1. Структуриране на програмния текст по блокове според използваните конструкти за управление. Текстът се огражда с правоъгълници, докато не се получи блоковата структура.
2. Илюстриране на елементите на средата. Това изисква да се състави схема, на която елементите на средата са представени по подходящ начин.
3. Съставяне на илюстрация на алгоритмичните преобразования при паралелно внимателно четене на текстовото описание на алгоритъма и на разградения на блокове текст на програмата.
4. Определяне на алгоритмичния смисъл на всеки блок и всеки оператор от програмния текст, като се номерира или означи по някакъв друг начин съответствието му с илюстрацията на алгоритмичните преобразования.
5. Съставяне на схемата на управление.

Задача 2.

Да се състави схемата на управление на метода на мехурчето с прекъсване по "индикатор" за инверсии при използване на итеративен цикъл със следусловие и да се запише съответстващия програмен текст, ограден по блокове. Да се "проиграе" на ръка методът с примерни данни, при използване на илюстрацията на алгоритмичните преобразования. Да се състави подходяща таблица за трасиране на изпълнението на програмния текст и да се направи трасировка.

1.3 Алгоритми за претърсване

1.3.1 Претърсване при ненаредени данни

Често срещана задача е тази за претърсването на структура, в която са записани данни и един от начините за нейното решаване се базира на обхождане на структурата.

Да разгледаме конкретен вариант на задачата за претърсване и да направим анализ на алгоритмите, базирани на проверки при обхождане.

Дадено:	**Търси се:**
В масива A има n данни.	Дали стойността X се намира в масива (Да или Не).

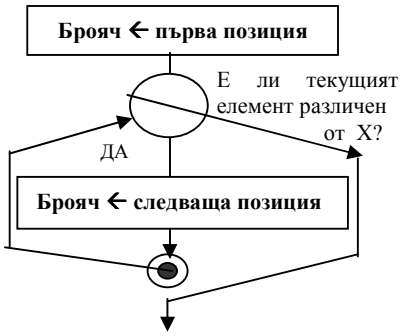
Очевидно решаването на тази задача с обхождане изисква най-много n проверки. Обхожда се масивът с въпроса "A[i] = x?".

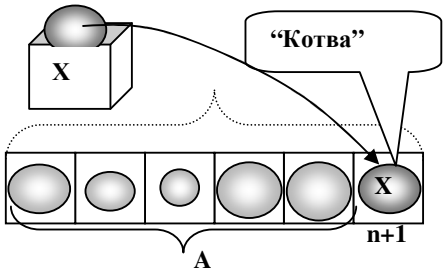
Ако се използва цикъл "по вграден брояч" с граници от първата до последната позиция от масива, проверките са винаги точно n.

При горната постановка на задачата могат и да се спестят сравнения. Нека обхождането спира с първото "намиране" на стойност от масива, равна на X. Тогава проверките са най-много n и най-малко 1. "Спирането при първото намиране" се реализира посредством итеративен цикъл и налага вграждане на

брояч, както е и показано на следващата схема. Това управление крие опасност за нарастване на брояча извън границите на масива, в случай че X не е намерен.

Ако няма никакви гаранции за това, че търсената стойност X е наистина записана някъде в масива, управлението трябва да се допълни с необходимите защити (проверка за текущата стойност на брояча). Това удвоява броя на извършваните проверки. В допълнение, за да се установи дали алгоритмичният блок с предлаганото долу управление е завършил, защото X е бил намерена, или защото масивът е бил обходен безрезултатно до края, трябва да се приложи схемата, която образно нарекохме "Защо излязох от цикъла?".





Ето и едно различно решение: прилага се една "хитрост", наричана "*котва*" (бариера). Търсената стойност X се поставя "насилствено" в масива, за да се гарантира, че тя ще бъде намерена и цикълът по единствено условие ще спре. За целта масивът се разширява с една позиция и в нея се записва X. Претърсването с "котва" има най-много n+1 проверки, но изисква също прилагане на схемата "Защо излязох?".

1.3.2 Претърсване по дихотомичен принцип

- ***Наредеността и дихотомия – принципна постановка***

Словесно описахме дихотомичния похват така: един обект се разделя на две части – такава, която отговаря на някакво условие и такава, която не отговаря на това условие. "Лошата" част се *отхвърля* и над останалата се прилага същата логика, със същия критерий. Не е задължително двете части да са равни. Просто – две части.

Разгледахме класическия метод "дихотомия" за решаване на уравнение. Прилагането на дихотомичен подход в алгоритми, работещи със структури от данни има специфика, на която в настоящата тема ще се спрем паралелно на изграждането на дихотомичен алгоритъм за работа с вече изградена структура.

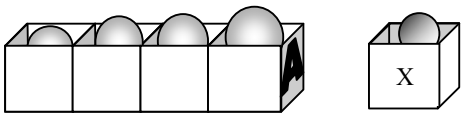
- ***Принципна постановка на дихотомичното претърсване***

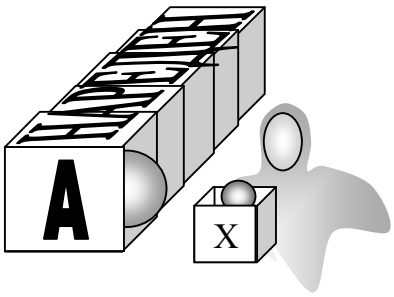
Дихотомичното претърсване на структура от данни *изисква* в данните в структурата *да са наредени.*

Да се спрем на конкретна ситуация на претърсване:

Даден е: масив A с данни, *наредени* във възходящ ред.

Търси се: дали стойността X е в масива – Да или Не.





Има ли смисъл да се обхожда, когато стойностите в масива са наредени по големина? Да разсъждаваме по принцип, като разгледаме два гранични случая: Ако търсената стойност X е по-малка от първата стойност на масива, няма как X да бъде намерена "по-нататък" в масива. Аналогично, ако X е по-голяма от последната стойност в масива, не е възможно да се намира на предходните позиции. И в двата разгледани случая отговорът на въпроса от условието на задачата е "*Не*" и задачата е решена.

Стремежът е да бъде изграден ефективен алгоритъм, т.е. – следи се за броя на проверките. Вижда се, че при наредени данни една единствена проверка може да доведе до отхвърляне на цял масив и да спести усилията по обхождане със задаване на безсмислени въпроси. Именно по този начин *наредбата помага на претърсването*. Тя прави възможно *отхвърлянето* на тези структурно организирани данни, за които сравненията са излишни. Наредбата е пръв помощник на претърсването. В този конкретен случай "една проверка – *отхвърля* масив".

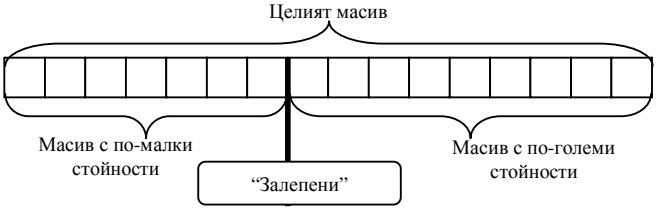
В хода на приложените горе разсъждения не става ясно какво да се прави, ако се окаже, че стойността на първия елемент на A *не е* по-голяма от стойността X. Идеята "следващата проверка да е за другия край на масива" е лоша, защото би довела също до обхождане, но в две противоположни посоки. Стремежът е да се приложи горното изключително ефективно разсъждение: една проверка – отхвърляне на масив.

Второто правило, което важи при прилагането на дихотомичен похват, е, че *трябва структурата да е представима като две подструктури*. При това, двете подструктури да не се различават принципно от цялата структура, да са изградени точно като нея и взети заедно, да я образуват. Тази логическа стъпка изглежда невинна, но когато се съставя дихотомичен алгоритъм тя е съществена за формализирането на задачата.

Всеки масив с повече от един елемент всъщност представлява два под-масива. Тук се работи с наредени масиви и затова двата под-масива са: "масив с по-малки стойности" и "масив с по-големи стойности".

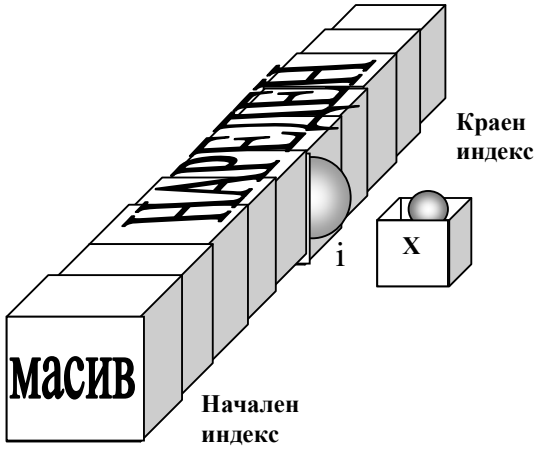
Да обърнем внимание на това, че нареденият масив може да се дефинира рекурсивно по следния начин:

Масив, това е:
1. Един елемент.
2. Два *масива* (залепени).



Наличието на рекурсивно дефиниран обект е добра изходна позиция за съставяне на рекурсивна процедура за решаване на задачата.

Сега ще изложим логиката на итеративния алгоритъм. Да обърнем внимание, че надолу разсъжденията не са конкретно за входния масив A, а за "*един нареден масив*", фиксиран от два индекса – "начален индекс" и "краен индекс".

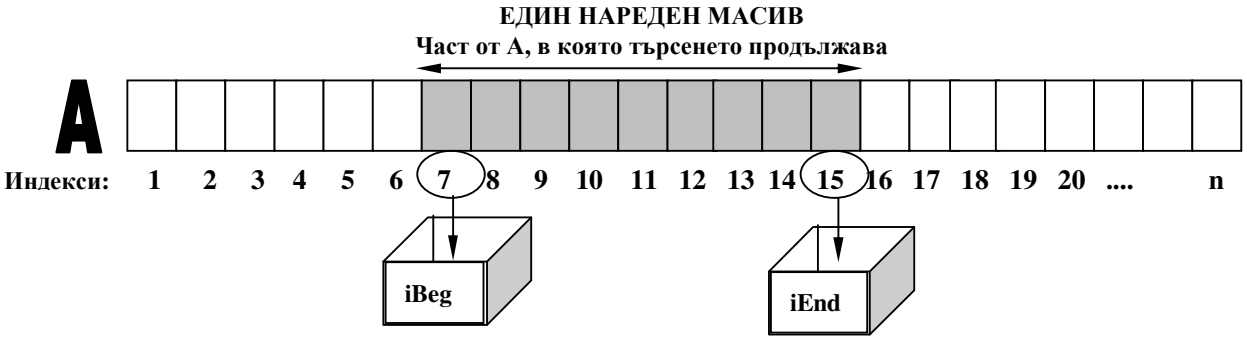


Произволен "некраен" индекс i разделя *един нареден масив* на два други наредени масива – масив в който стойностите на елементите са по-големи или равни от стойността в позиция [i] и масив, в който стойностите на елементите са по-малки или равни на нея. По илюстрацията горе може да се състави алгоритъм за разглежданата задача, като се използват направо отговорите на следните три въпроса:

- Какъв е изходът, т.е. отговорът на условието на задачата, ако стойността от позиция [i] е равна на X?
- Ако стойността от позиция [i] е по-голяма от стойността X, коя част на масива да се *отхвърли*?
- Кои са първият и последният индекси на масива, който не е отхвърлен и остава за по-нататъшно претърсване?

Очевидно, както при дихотомичното решаване на уравнение, масивът, който остава за претърсване, "става" все по-малък и по-малък. По принцип, процесът на претърсване се развива в масива A, като алгоритъмът "отделя" негови под-масиви. За да се индексира някой под-масив на A, може де се използва кое да е индексно множество от ординален тип.

В тази конкретна задача индексите на под-масива остават такива, каквито са индексите на A.



Нека за граничните индекси на под-масив на A въведем променливите iBeg и iEnd. Тези променливи съдържат индексите на A, които определят в коя негова част претърсването продължава. Сега можем да формулираме задачата на алгоритъма по следния начин: алгоритъмът мести граничните индекси iBeg и iEnd на масива, определен горе с термина "един нареден масив".

Третото правило, което трябва да се спазва при прилагане на дихотомично отхвърляне, е, че процесът трябва да борави с такава "субстанция", намаляването на която има гарантиран край. Тогава алгоритъмът със сигурност ще *завършва.* При прилагането на дихотомично отхвърляне над структури от данни това лесно се реализира, защото е достатъчно процесът на отхвърляне да довежда до получаването "*градивен атом*" на структурата, т.е. до състояние, в което последната останала за "обработване" подструктура просто не може повече да се дели.

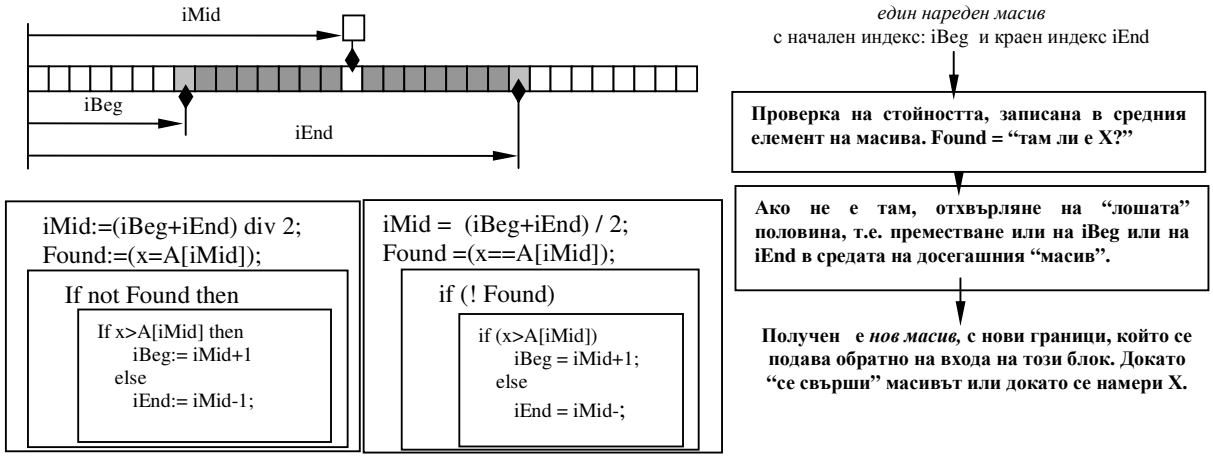
В случая става точно така – "*един нареден масив*" се дели на два под-масива, от които единният се отхвърля, а другият се подлага на същата обработка. Този процес има *гарантиран край*, дори търсената стойност X да не е записана никъде във входния масив. Все някога останалото за обработка ще бъде масив от един елемент и тогава алгоритъмът трябва да спре, защото масивът "се е свършил". Изчерпила се е структурата.

Ако се проследи рекурсивната дефиниция, неявно използвана тук, установява се, че самата дефиниция не позволява безкраен процес на "намаляване" на структурата, защото се базира на градивния атом – един елемент.

Остана да коментираме *как* по-точно да се раздели масивът на две части. Обикновено спонтанно се предлага масивът да се раздели на две *равни части*. Такова разпределение на големината на входа се нарича *идеален баланс*. На всяка поредна стъпка не е известно коя част ще бъде отхвърлена. Балансът е основна концепция в структури от данни, алгоритмите над които са особено "производителни" именно заради "разпределянето на бройката" по коментирания начин.

На следващата фигура е показан алгоритмичен блок, който реализира описаното дихотомично претърсване. В примерния програмен текст са използвани две помощни променливи – една за индекса на средния елемент на масива и една булева променлива-индикатор. Очевидно в цялостния алгоритъм за дихотомично претърсване този блок трябва да се надгради с итеративен

цикъл. Цикълът следва да завършва или при намерено X, или при "изчерпан" масив. Това налага добавянето на блок "Защо излязох от цикъла?".

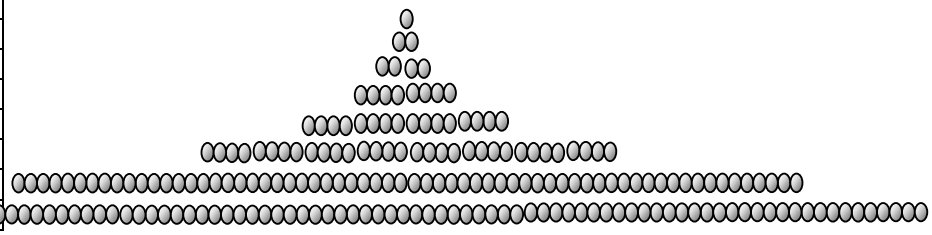


➢ *Елементи на анализ на сложност на алгоритъма за дихотомично претърсване на масив*

Анализът на сложността следва да даде отговор на въпроса: "Ако на входа на алгоритъма постъпи нареден масив A с n елемента, след колко стъпки алгоритъмът ще си е свършил работата?"

Нека най-напред разгледаме схема, даваща нагледна представа за скоростта, с която се обработва входът при дихотомично отхвърляне. Да си представим, че има n елемента, в които са записани стойности, които подлежат на претърсване. В термините, с които дефинирахме структурата "масив" при разработването на алгоритъма, дадени са n "атомчета". Алгоритъмът спира най-късно след като "стигне до масив-атом".

Ниво	Брой	
0	1	2^0
1	2	2^1
2	4	2^2
3	8	2^3
4	16	2^4
5	32	2^5
6	64	2^6



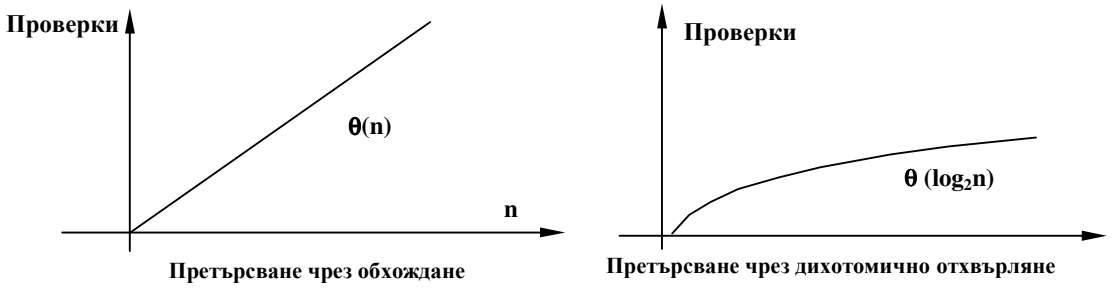
На схемата горе е изобразена геометрична прогресия със знаменател 2. На всяко следващо ниво има два пъти повече елементи от тези на предходното ниво. Да си представим, че най-долу, в основата на тази "постройка", стои входът на задачата, състоящ се от $n=2^k$ елемента. А сега да си представим, че един алгоритъм, който претърсва чрез обхождане ще "се придвижва" елемент по елемент в хоризонтално направление, до края на реда. А алгоритъм, който претърсва дихотомично, ще се придвижва във вертикално направление, най-

много до върха на постройката. Тази схема на обработване на входа е основна схема за работа алгоритми, квалифицирани с право като *"най-ефективни"*.

Входен масив от 2^k елемента може да се претърси с най-много k проверки. Независимо от това дали големината n на входа е точна степен на 2, можем да си представим, че той е увеличен допълнително до най-близкото 2^k. Дори така проверките са *само k*. Очевидно при тази постановка броят сравнения *k* е горна цяла част на $log_2 n$.

$k = \lceil \log_2 n \rceil$

Това е и функцията-еталон за оценката на алгоритъма – сложността му е $\theta(\log_2 n)$.



Долу са дадени илюстрации за работата на метода при претърсването на масив от 14 елемента, веднъж за стойност, налична в масива, и веднъж за стойност, неналична в масива. Проследете илюстрациите успоредно с програмния текст и трасирайте преместването на индексите и резултатите от проверките на всяка стъпка от алгоритъма.

X=17

i=1	i=2	i=3	i=4	i=5	i=6	i=7	i=8	i=9	i=10	i=11	i=12	i=13	i=14
2	4	5	7	8	10	13	15	17	18	21	21	22	27
iBeg						iMid							iEnd
2	4	5	7	8	10	13	15	17	18	21	21	22	27
							iBeg			iMid			iEnd
2	4	5	7	8	10	13	15	17	18	21	21	22	27
							iBeg	iMid	iEnd				

X=9

i=1	i=2	i=3	i=4	i=5	i=6	i=7	i=8	i=9	i=10	i=11	i=12	i=13	i=14
2	**4**	**5**	**7**	**8**	**10**	13	**15**	**17**	**18**	**21**	**21**	**22**	**27**
iBeg						iMid							iEnd
2	**4**	**5**	**7**	**8**	**10**	**13**	**15**	**17**	**18**	**21**	**21**	**22**	**27**
iBeg		iMid			iEnd								
2	**4**	**5**	**7**	**8**	**10**	**13**	**15**	**17**	**18**	**21**	**21**	**22**	**27**
			iBeg	iMid	iEnd								
2	**4**	**5**	**7**	**8**	**10**	**13**	**15**	**17**	**18**	**21**	**21**	**22**	**27**
					iBeg								
					iEnd								

- ***Кратки разсъждения и идеи за подобрения на оценката на алгоритъм***

Вдясно е дадена схема на блоковете на най-високото ниво на детайлизация на алгоритъм за претърсване на масив от n данни. Както се вижда от схемата, усилията, които този алгоритъм полага, за да установи дали X е в масива, включват усилията за начално сортиране на масива.

Общата оценка за сложността по време на този алгоритъм е сума от тази за сортирането на масива и тази за претърсването му. Сортирането е "по-трудоемък" процес и предопределя вида на общата оценка.

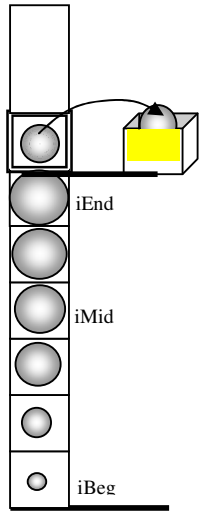
Например, нека сортирането става с някой от разгледаните дотук алгоритми, които извършват от порядъка на n^2 сравнения. Поведението на горния алгоритъм би било "като $n^2 + log_2 n$ ". Най-силно то се предопределя от начина на нарастване на квадратната функция и следователно е отново " като $n^{2.}$ ".

Въпросът е дали си заслужава да се сортира най-напред, само и само да се облекчи претърсването, когато може масивът да се обходи без да е нареждан и това да става с най-много n сравнения. Отговорът на този въпрос зависи от ситуацията, в която се налага да се прилага илюстрираният на схемата алгоритъм. Ако данните "пристигат" еднократно и впоследствие се налага да бъдат претърсвани много пъти, предварителното им нареждане "си заслужава" труда. В този случай блокът за дихотомично претърсване ще е оформен като

процедура, която се повиква многократно от главната програма. Помислете как да се организира обменът на информация на такава процедура с главната програма.

Разбира се, много добре би било да се подобри оценката за сложност на алгоритъма от схемата горе, като се подобри оценката на блока "сортиране". Ще подскажем начин за подобряване на тази оценка при използване само на изложените дотук алгоритми. Както споменахме при разглеждането на алгоритъма "пряко вмъкване", той се поддава на значително подобрение поради това, че търсенето на място за вмъкване става в нареден под-масив. Ако анализираме по-внимателно постановката на претърсването ще открием, че няма особено значение дали се търси мястото за вмъкване или се търси стойността X.

Ще използваме дадените тук означения и от двата алгоритъма, за да представим най-общо идеята. Да разсъждаваме по принцип за търсенето на "правилно" място за вмъкваната стойност на базата на илюстрацията вдясно. (Ако стойността е равна на средния елемент от наредения под-масив, търсената позиция е точно след него, отгоре. Мястото е намерено. А ако не е равна?) Отново основна е логиката на *отхвърлянето*. В контекста на търсенето на правилно място, тя се интерпретира така: "правилното място на този елемент *в никакъв случай не е* в този под-масив!".



Състояние I

Това разсъждение подсказва, че след като постепенно са отхвърлени всички получавани (все по-малки и по-малки) под-масиви и проверяваният под-масив е останал с единствен елемент, "вече е време за вмъкване". Останалото за изпълнение е освобождаването на правилната позиция с преместване на стойности от наредения под-масив нагоре.

При тази постановка вмъкването става с логаритмична сложност по време за всички $n-1$ последователни състояния на масива. Общо, броят на проверките става от порядъка на $n \cdot log_2 n$. Това съществено подобрение на алгоритъма на прякото вмъкване го прави сравним по показателя "брой проверки" с най-добрите алгоритми за сортиране. Това съпоставяне на получения общ положителен ефект подчертава голямото значение на работата с наредени структури за добрата производителност на алгоритмите.

♦ *Примерни задачи за упражнение*

Задача 1.

Да се състави алгоритъм и приграма за дихотомично претърсване на нареден масив.

Задача 2.

Да се състави подробна илюстрация на алгоритмичните преобразования за алгоритъма за сортиране “пряко вмъкване” при използване на дихотомично търсане на мястото на вмъкване. Да се състави схема на управление и примерен програмен текст.

Съвет

Проследете внимателно схемата на дихотомичното претърсване на нареден масив когато “елементът не е намерен”. Къде са индексите за начало и край на претърсвания масив след последната стъпка на алгоритъма?

2 ПРИМЕРИ

2.1 Предговор от съставителя на примерите

Много от добрите книги по алгоритми ползуват псевдокод вместо конкретен език за програмиране. Това е обяснимо – специфичните особености на езика и детайлите на синтаксиса често пречат идеята на алгоритъма да бъде изложена кратко и ясно. От друга страна, едно учебно пособие по програмиране би загубило значителна част от своята практическа полезност, ако в него отсъствуват примерни текстове на програми, съставени на използваем език за програмиране. Настоящият учебник, в качеството си на уводен в алгоритмите и програмирането е съставен с отчитане на това противоречие. В основното изложение, ориентирано към алгоритми и тяхната процедурна реализация използването на програмен текст е сведено до минимум. Примерните програмни текстове са оформени като самостоятелна част на учебника, но тяхната последователност е в съответствие с основното изложение. Примерите са групирани в 11 теми в последователност, следваща предложените в изложението задачи за самостоятелна работа. Във всяка тема се предлага програмен код за решение на задачи, сродни с поставените или на техни варианти. С това обучаемите се ориентират към похвати за кодиране, подходящи за поставената задача.

Езикът Pascal се използува традиционно в уводни курсове по програмиране. В съвременната масова програмистка практика обаче преобладават езиците със C-подобен синтаксис. Действуващият стандарт на C++ предлага едно подмножество на езика, което е напълно съпоставимо с Pascal по отношение на диагностика и устойчивост срещу “грешки на начинаещия”. Примерите в настоящия учебник са разработени като конзолни приложения паралелно в две езикови среди: Borland Delphi и Microsoft Visual C++. Изборът е предопределен от наличието на такъв лицензиран софтуер в НБУ. Простотата на приложените примери и текстовият вход-изход осигуряват безпроблемната им обработка от компилатори на други производители. Предварителна подготовка по един от двата езика се предполага, но разглежданите примери позволяват усвояване на езика паралелно с курса.

В предлаганите програмни текстове съзнателно е търсено максимално съответствие между кода на Pascal и този на C++ за една и съща задача. Всъщност, C++ е използван вместо C и заради разширения синтаксис, позволяващ този паралелизъм да бъде следван. На читателите, добре усвоили синтаксиса на единия език се препоръчва да преглеждат и “паралелния” код. И пасивното ползуване (четене и разбиране) на още един език е полезно, както при естествените езици, така и в програмирането. Предназначението на тази част на учебника е не толкова да предостави изходния код на действуващи програми (достъп до този код съществува през сайта на НБУ), колкото да създаде навици за четене на програмен код. За тази цел в кода обилно са вложени коментари. Нека тези коментари да не предизвикват досада у читателя. Нищо че са в излишък, като за начинаещи. Навиците за добро документиране

на програмите, в това число и чрез коментари, трябва да се създават своевременно.

Проблемът с възможна разлика в кирилизацията на средата за програмиране и произведеното с нея конзолно приложение доведе до решението всички извеждани от програмите съобщения да бъдат написани на латиница. Вместо дразнещата (и общо взето произволна) транскрипция на български текст с латиница беше предпочетено съобщенията да са на английски. Читателят може да промени тези съобщения по свое виждане, ръководейки се от написаното на български в коментарите.

2.2 Общи характеристики на текста на програма на процедурен език от високо ниво

Примерите илюстрират общите характеристики на програмния текст в използваните среди за програмиране.

Коментарът е средство за вмъкване в програмния текст на пояснения за читателя, които не се разглеждат от компилатора, т.е. не са част от програмата. И в двете среди текстът от // до края на реда може да служи за коментар. По-дълги коментари се ограничават с { } или (* *) в Pascal и с /* */ в C.

Елемент от текста, който не се компилира, а определя условията и параметрите на компилацията са така наречените *директиви.* В Pascal директивите се ограждат с { }, като първият знак след { е $. В C директивите започват с # и се изписват на отделен ред.

Използуването на готови библиотеки се предписва в C с директивата **#include**, а в Pascal – чрез служебната дума **uses** в отделен оператор от програмата.

Кодът на програмата се състои от блокове, блоковете – от оператори (с възможно влагане на други блокове), а операторите - от лексеми. Лексемите са най-малките текстови единици със самостоятелен смисъл и биват: служебни думи (**if**, **while** и т.н.); разделители (; , +, - и т.н.); символични имена (на променливи, типове, процедури и пр.); константи (напр. числови като 5.0). Символичните имена не могат да съвпадат със служебна дума и не бива да започват с разделител или със символ, допустим като начален за константа. Навсякъде в текста на програмата между две лексеми може да бъде вмъкнат коментар, произволен брой интервали или знак за край на ред. В Pascal не се прави разлика между главни и малки букви, в C тази разлика се отчита.

Приликите и разликите в синтаксиса на двете езикови среди се виждат в следващите примери, където отделните елементи на текста са коментирани.

```
// Програмен текст на Pascal.
// Среда (IDE): Borland Delphi
// Това, което се чете на български са коментари
{ и това е коментар } (* и това също *)
// Коментарите не се превеждат в изпълним код.
```

```
// Заглавие на програмата. Може да се изпусне, но ако съществува,
// името на програмата (hello1) не може да се използува вътре в текста
// за други цели.
program hello1;

// Следва директива за типа на изпълнимия код:
// конзолно приложение, т.е. програма с текстов вход/изход.(само за Delphi)
{$APPTYPE CONSOLE}

uses SysUtils; //декларация на използувани библиотеки

// Тук се поставят декларациите на типове и променливи.
// В този текст такива липсват.

//Следва тялото на програмата, ограничено от
begin                              //служебна дума за отваряща скоба на блок
  write('Hello, today is ');       //извеждане ("отпечатване")
                                   //на константа-символен низ
  writeln(DateTimeToStr(Date));    //извеждане на резултат от обръщение към
                                   //функция: DateTimeToStr преобразува в текстов
                                   //вид системната дата,получена от функция Date
  write('Press <Enter> to close');//извеждане на подсещане за потребителя
  readln;                          //вход от потребителя – натискане на <Enter>
end.                               //служебна дума за затваряща скоба на блок
                                   //програмният текст завършва с точка.


// Програмен текст на C++
// Това, което се чете на български са коментари
/* и това е коментар */
// Коментарите не се превеждат в изпълним код.
// Следват директиви за включване на библиотеки
#include <iostream.h> //C++ библиотека за вход/изход
#include <time.h>     //C библиотека за дата/време
// Директивите определят "обкръжението", в което се
// обработва програмата от езиковата среда (C++ IDE).

// Следва текст на програмата
// Функцията  с име main поема управлението при стартиране на
// изпълнимия код на програмата.

void main()                    //заглавие на функцията
// Тяло на функцията, ограничава се от:
{                              //отваряща скоба на блок
 char tmpbuf[128];             //декларация на променлива
 _strdate( tmpbuf );           //обръщение към функция
 cout<<"Hello! Today is ";     //извеждане ("отпечатване")
                               //на константа-символен низ
 cout<< tmpbuf <<endl;         //извеждане на променлива
}                              //затваряща скоба на блок



// Програмен текст на Pascal.
// В този програмен текст има променлива- символен низ, в която се въвежда име
// на потребителя.
uses SysUtils; //декларация на използувани библиотеки

// Тук се поставят декларациите на типове и променливи.
var name:string;
//Следва тялото на програмата.
begin
  write('Hello, what is your name?');  //извеждане ("отпечатване")
                                       //на константа-символен низ
```

```
  readln(name);                          //въвеждане на името
  writeln('Hello, ', name,'!');          //извеждане на поздравление
  write('Today is ');                    //извеждане на съобщение за
  writeln(DateTimeToStr(Date));          //текущата дата
  write('Press <Enter> to close');
  readln;
end.
```

2.3 Езикова реализация на конструкти за управление

Проста програма с употребата на условен оператор if се предлага като първи пример за употреба на структури за управление. Проверката за делимост на цели числа е необходима за алгоритъма на Евклид, разработван впоследствие.

```
//Проверка за делимост на две въведени цели числа
program Divider3;
{$APPTYPE CONSOLE}
uses  SysUtils;
var a,b:integer;
begin
 writeln('Divisibility of integers.');      // За какво служи програмата.
 write('Enter two integers: ');             // Подсещане за потребителя.
 readln(a,b);                               // Въвеждане на данни.
 if a mod b =0 then                         // Ако има делимост
  writeln(b,' is divider of ',a)            // се отпечатва съобщение,
 else                                       // иначе
  writeln(b,' is not divider of ',a);       // се отпечатва друго съобщение.
  writeln('Press <Enter> to close.');       // Подсещане.
 readln;                                    // Четене на потребителски вход.
end.                                        // Край.

//Проверка за делимост на две въведени цели числа
#include <iostream.h>
void main()
{int a,b;
 cout<<"Divisibility of integers."<<endl;    // За какво служи програмата.
 cout<<"'Enter two integers: ";              // Подсещане за потребителя.
 cin>>a>>b;                                  // Въвеждане на данни.
 if(a%b==0)                                  // Ако има делимост
  cout<<b<<" is divider of "<<a<<endl;       // се отпечатва съобщение,
 else                                        // иначе
  cout<<b<<" is not divider of "<<a<<endl;   // се отпечатва друго съобщение.
}
```

Предлагат се няколко варианта на реализация на алгоритъма на Евклид, в които повторенията са организирани с различни оператори за управление: условен оператор + безусловен оператор за преход; цикъл с предусловие; цикъл със следусловие. Употребата на етикети и оператор **goto** не се препоръчва, предложена е тук като допълнително пояснение за действието на операторите за цикъл.

```
// Алгоритъм на Евклид, реализиран "пряко" – с изход от средата на цикъла
program Cycles;
{$APPTYPE CONSOLE}
uses  SysUtils;
// За делимо, делител и остатък се ползуват променливите i,j,k.
// Входните данни се въвеждат в a и b.
```

```
var i,j,k,a,b:integer;
// За организиране на цикъла са необходими два етикета
// В Pascal те се описват предварително чрез служебната дума label.
label again,stop;
begin
  write('Enter two integers to find their ',
        'greatest common divisor:');
  readln(a,b);
  i:=a;j:=b;

// Оттук започват операторите, представящи алгоринъма
again:                  //начало на цикъла
 k:=i mod j;
 if k=0 then goto stop  //прекратява се, ако е намерено решение
 else                   //иначе
   begin
    i:=j;j:=k;          //смяна на стойностите на делимото и делителя
    goto again;         //повторение
   end;
// Тук се предава управлението при изпълнено условие k=0
 stop: writeln('The GCD of ',a,' and ',b,' is ',j);
// Край на алгоритъма
 i:=a;j:=b; //Подготовка за ново изпълнение на алгоритъма, този път представен
//    чрез оператор за цикъл и обръщение към функция за прекратяване на цикъла
// Изход от средата на цикъла може да се организира в Delphi чрез обръщение
// към процедурата break за прекратяване изпълнението на цикъл.
 repeat
   k:=i mod j;
   if k=0 then break;
   i:=j;j:=k;
 until false;             //без break този цикъл е безкраен
 writeln('The GCD of ',a,' and ',b,' is ',j);
 writeln('Press <Enter> to continue.');
 readln;
end.

// Алгоритъм на Евклид, реализиран чрез цикъл със "следусловие"
program cycles2;
{$APPTYPE CONSOLE}
uses  SysUtils;
// За делимо, делител и остатък се ползуват променливите i,j,k.
// Входните данни се въвеждат в a и b.
var i,j,k,a,b:integer;
// За организиране на цикъл без оператори repeat-until е необходим един етикет
label again;
begin
  write('Enter two integers to find their ',
        'greatest common divisor:');
  readln(a,b);
  i:=a;j:=b;                      //подготовка за алгоритъма на Евклид

// Реализация на цикъла без оператори repeat-until
  again:                          //вход в цикъла
    k:=i mod j;                   //пресмятане на остатъка
    i:=j;j:=k;                    //подготовка за ново пресмятане (XX)
                                  //независимо от това дали е намерено решение
  if k<>0 then goto again;        //повторение, ако k<>0
  // Тук се достига при намерено решение.
  // Поради направените присвоявания (XX), последната стойност на делителя се
  // намира в i.
  writeln('The GCD of ',a,' and ',b,' is ',i);
// Край на изпълнението.

  i:=a;j:=b;
```

```
// Текстът оттук до последния коментар е функционално равностоен на горния
// Цикъл със "следусловие" в Pascal се представя с repeat-until
  repeat        // маркира началото на тялото на цикъла
    k:=i mod j;
    i:=j;j:=k;
  until k=0;    // маркира края на тялото на цикъла
                //и съдържа условието за КРАЙ НА ЦИКЪЛА (не за продължаване)
  writeln('The GCD of ',a,' and ',b,' is ',i);
// Край на алгоритъма на Евклид със следусловие, записан чрез repeat-until
  write('Press <Enter> to close.');
  readln;
end.
```

```
// Алгоритъм на Евклид, реализиран чрез цикъл с "предусловие"
program cycles1;
{$APPTYPE CONSOLE}
uses  SysUtils;
var i,j,k,a,b:integer;
// За организиране на цикъл без оператор while са необходими два етикета
label again,stop;
begin
  write('Enter two integers to find their ',
        'greatest common divisor:');
  readln(a,b);
  i:=a;j:=b;
// Реализация на цикъла без оператор while
 k:=i mod j;             //подготовка на предусловието
  again:                 //начало на цикъла
  if k=0 then goto stop  //проверка на предусловието
  else
   begin
    // това са изпълнимите оператори от тялото на цикъла,
    i:=j;j:=k;
    k:=i mod j;  //включващи промяна на проверяваната в условието стойност
    goto again;  //и накрая - безусловен преход към началото на цикъла
   end;
  stop:
  // Тук се преминава при намерено решение.
  writeln('The GCD of ',a,' and ',b,' is ',j);
  write('Press <Enter> to close.');
  readln;
end.
```

```
#include <iostream.h>
// Алгоритъм на Евклид, реализиран "пряко" - с изход от средата на цикъла
void main()
// За делимо, делител и остатък се ползуват променливите i,j,k.
// Входните данни се въвеждат в a и b.
{int i,j,k,a,b;
  cout<<"Enter two integers to find their "<<
        "greatest common denominator:"<<endl;
  cin>>a>>b;
  i=a;j=b;
// Оттук започват операторите, представящи алгоринъма
// За организиране на цикъла са необходими два етикета
```

```
again: //начало на цикъла
 k=i % j;
 if(k==0)goto stop; //прекратява се, ако е намерено решение
 else                  //иначе
 {i=j;j=k;          //смяна на стойностите на делимото и делителя
  goto again;         //повторение
 }
// Тук се предава управлението при изпълнено условие k=0
 stop: cout<<"The GCD of "<<a<<" and "<<b<<" is "<<j<<endl;
// Край на алгоритъма
 i=a;j=b;

// Изход от средата на цикъла може да се организира в C чрез
// оператора break за прекратяване изпълнението на цикъл.
 do
 {
   k=i % j;
   if (k==0)break;
   i=j;j=k;
 }while(true); //без break този цикъл е безкраен
 cout<<"The GCD of "<<a<<" and "<<b<<" is "<<j<<endl;
}

#include <iostream.h>
// Алгоритъм на Евклид, реализиран чрез цикъл със "следусловие"
void main()
// За делимо, делител и остатък се ползуват променливите i,j,k.
// Входните данни се въвеждат в a и b.
{int i,j,k,a,b;
  cout<<"Enter two integers to find their "<<
        "greatest common denominator:"<<endl;
  cin>>a>>b;
  i=a;j=b;
// Оттук започват операторите, представящи алгоринъма
// За организиране на цикъл без оператор do-while е необходим един етикет
// Реализация на цикъла без do-while
  again:                      //вход в цикъла
    k=i % j;                  //пресмятане на остатъка
    i=j;j=k;                  //подготовка за ново пресмятане (XX)
                              //независимо от това дали е намерено решение
  if (k!=0)goto again;        //повторение, ако k!=0
  // Тук се достига при намерено решение.
  // Поради направените присвоявания (XX), последната стойност на делителя
  //се намира в i.
  cout<<"The GCD of "<<a<<" and "<<b<<" is "<<i<<endl;
// Край на изпълнението.

  i=a;j=b;

// Текстът оттук до последния коментар е функционално равностоен на горния
// Цикъл със "следусловие" в C се представя с do-while
  do{        // маркира началото на тялото на цикъла
    k=i % j;
    i=j;j=k;
  }while(k!=0);    // маркира края на тялото на цикъла
                  //и съдържа условието за ПРОДЪЛЖАВАНЕ НА ЦИКЪЛА
  cout<<"The GCD of "<<a<<" and "<<b<<" is "<<i<<endl;
// Край на алгоритъма на Евклид със следусловие, записан чрез do-while
}

#include <iostream.h>
// Алгоритъм на Евклид, реализиран чрез цикъл с "предусловие"
void main()
// За делимо, делител и остатък се ползуват променливите i,j,k.
```

```
// Входните данни се въвеждат в a и b.
{int i,j,k,a,b;
  cout<<"Enter two integers to find their "<<
        "greatest common denominator:"<<endl;
  cin>>a>>b;
  i=a;j=b;
// Оттук започват операторите, представящи алгоринъма
// За организиране на цикъл без оператор while са необходими два етикета
// Реализация на цикъла без оператор while
 k=i %j;                  //подготовка на предусловието
  again:                  //начало на цикъла
  if (k==0)goto stop;     //проверка на предусловието
  else
  {// това са изпълнимите оператори от тялото на цикъла,
    i=j;j=k;
    k=i % j;      //включващи промяна на проверяваната в условието стойност
    goto again;  //и накрая – безусловен преход към началото на цикъла
  }
  stop:
  // Тук се преминава при намерено решение.
  cout<<"The GCD of "<<a<<" and "<<b<<" is "<<j<<endl;
// Край на алгоритъма

}
```

2.4 Целочислени алгоритми с циклична структура

Решението на следващата задача има организация, подобна на алгоритъма на Евклид. В качеството на упражнение може да се пробват различни реализации на цикъла, както бе направено по-горе.

```
//Многократна делимост на цели числа
program MDivider3;
{$APPTYPE CONSOLE}
uses SysUtils;
// Задачата е да се провери дали едно число е
// многократен делител на друго, т.е. делители
// да са негови цели степени b^c, c=1,2..

// Делимо a, делител b, брояч c, начална стойност на
// делимото a1.
var a,b,c,a1:integer;
begin
 writeln('Multiple divisibility of integers.');
 write('Enter the dividend: ');
 readln(a);
 repeat                       //въвеждане на делителя
  write('Enter the divider: ');
  readln(b);
 until abs(b)>1;              //със защита от стойности 0 и 1

 c:=0;                        //инициализиране на брояча
 a1:=a;                       //запазване на началната стойност
 while a mod b=0 do           //докато има делимост
 begin
  c:=c+1;                     //нарастване на брояча
  a:=a div b;                 //подготвя се продължение с
                              //резултата от делението
 end;
 if c =0 then                 //не е регистрирана делимост
  writeln(b,' does not divide ',a1)
 else if c=1 then             //еднократна делимост
```

```
  writeln(b,' is divider of ',a1)
 else                          //многократна делимост
 writeln(b,' is divider of ',a1,
   ' more than once: ',c,' times.');
 writeln('Press <Enter> to close.');
 readln;
end.
```

```
//Многократна делимост на цели числа
#include <iostream.h>
// Задачата е да се провери дали едно число е
// многократен делител на друго, т.е. делители
// да са негови цели степени b^c, c=1,2..
void main()
{
// Делимо a, делител b, брояч c, начална стойност на
// делимото a1.
 int a,b,c=0,a1;
 cout<<"Multiple divisibility of integers."<<endl;
 cout<<"Enter the dividend: ";
 cin>>a;
 do                            //въвеждане на делителя
 {cout<<"Enter the divider: ";
  cin>>b;
 }while(abs(b)<=1);            //със защита от стойности 0 и 1
 a1=a;                         //запазване на началната стойност
 while (a%b==0)                //докато има делимост
 {c++;                         //нарастване на брояча
  a=a/b;                       //подготвя се продължение с
                               //резултата от делението
 }
 if(c==0)                      //не е регистрирана делимост
  cout<<b<<" is not divider of "<<a1<<endl;
 else if (c==1)                //еднократна делимост
  cout<<b<<" is divider of "<<a1<<endl;
 else                          //многократна делимост
  cout<<b<<" is divider of "<<a1
  <<" more than once: "<<c<<" times."<<endl;
}
```

2.5 Коректност на цикличното управление.

Приведеният пример илюстрира възможността привидно правилно конструиран цикъл да не завърши или за завърши с непредсказуем резултат. Алгоритъмът на Евклид, изпълнен над реални числа може да бъде наречен “най-голяма обща мярка (НОМ) на две отсечки ” . Както е известно, такава мярка не съществува за всеки две отсечки. Например, катетът в равнобедрен правоъгълен триъгълник не е съизмерим (няма НОМ) с хипотенузата. Поради крайното (и рационално) представяне на реалните числа в компютъра приложеният програмен текст ще дава решение и в случаите, когато изходната математическа задача няма решение. Изведеният резултат е едно малко число, чиято стойност зависи от входните данни, но и от конкретното представяна на числа с плаваща запетая. Подобен е резултатът и при двоично представяне на рационални числа с недостиг. Например, десетичната дроб 0.1 се представя като безкрайна периодична двоична дроб, чието представяне в компютъра се

ограничава от дължината на машинната дума. Поради това, за НОМ на 1 и 0.1 няма да се получи верен резултат. Влиянието на точността на представяне може да се изследва, като се изпробва различна дължина на машинната дума за променливите a, b и r. В текста на Pascal пробвайте single и extended вместо real, в текста на C – double вместо float.

```
program NOM3;
{$APPTYPE CONSOLE}
uses  SysUtils;
var a,b,r:real;
begin
 writeln('Greatest common "measure" of two floats.');
 write('Enter two real numbes: ');
 readln(a,b);
 writeln('The common measure of ');
 write(a:15:6,' and ',b:15:6,' is ');
 r:=a-b*trunc(a/b);
 while r<>0 do
 begin
  a:=b;b:=r;
  r:=a-b*trunc(a/b);
 end;
 writeln(b);
 writeln('Press <Enter> to close.');
 readln;
end.
```

```
#include <iostream.h>
void main()
{float a,b,r;
 cout<<
 "Greatest common 'measure' of two floats."<<endl;
 cout<<"Enter two float numbes: ";
 cin>>a>>b;
 cout<<"The common measure of "<<a<<" and "<<b<<" is ";
 r=a-b*int(a/b);
 while(r!=0)
 {a=b;b=r;
  r=a-b*int(a/b);
 }
 cout<<b<<endl;
}
```

Следващият текст илюстрира риска от цикъл, прекратяван от проверка за равенство. Явно е, че при увеличаване на променливата на цикъла търсената стойност ще бъде прескочена. Ако програмата все пак завършва, то е защото настъпва целочислено препълване и обхождането на стойности продължава, като при препълването се улучва (случайно) цяло отрицателно число, чиято разлика с търсеното (35) е кратна на 3. Изведената на печат стойност на променливата с показва броя на изпълнените стъпки на цикъла (при 32-битови integer над 1.4 милиарда стъпки вместо около 10).

```
program Project2;
{$APPTYPE CONSOLE}
uses  SysUtils;
var c:longint;a:integer;
begin
```

```
 c:=0;a:=3;
 while a<>35 do
 begin
  a:=a+3;c:=c+1;
 end;
 writeln(c,' ',a);
 readln;
end.
```

```
#include <iostream.h>
void main()
{int a; unsigned int c;
 for (a=3,c=0;a!=35;a+=3)c++;
 cout<<c<<' '<<a<<endl;
}
```

2.6 Скаларни типове данни

Скаларните типове данни са представени с реализация на изброим тип в двете езикови среди. Работа със символния тип char е показана в раздел 8. Функции.

```
//Реализация на изброим тип в Pascal
program Enums4;
{$APPTYPE CONSOLE}
uses SysUtils;
// Чрез дефинирането на изброим тип на последователност
// от цели числа се задават имената, изброени в().
// В случая, sun=0,mon=1 и т.н..
// Името Days е име на така създадения тип.
type Days=(sun,mon,tue,wed,thi,fri,sat);

// Стойностите на променливи от тип Days не се преобразуват
// автоматично в имената им при отпечатване чрез write.
// Ако се желае отпечатването на съответните числа: 0, 1 ...,
// трябва да се използува функцията ord.
// Ако се желае отпечатване на имената, последните трябва
// да бъдат описани чрез символни низове.
// Това е направено чрез следващата декларация:
const DNames:array[Days]of string[3]=
  ('Sun','Mon','Tue','Wed','Thu','Fri','Sat');
//Програмата демонстрира разпечатване на номерата и имената
//на дните от седмицата в права и обратна последователност.
//Използуван е цикъл for с управляваща променлива от тип Days.

var day:Days;
begin
 writeln('Weekdays enumeration:');
 for day:=sun to sat do writeln(ord(day),' ',DNames[day]);
// След края на цикъла променливата day има стойност
// ИЗВЪН ДЕФИНИЦИОННАТА ОБЛАСТ!!!
 writeln(ord(day));
 writeln('Weekdays countdown:');
 for day:=sat downto sun do writeln(ord(day),' ',DNames[day]);
// След края на цикъла променливата day има стойност
// ИЗВЪН ДЕФИНИЦИОННАТА ОБЛАСТ!!!
 writeln(ord(day));
 writeln('Press <Enter> to close.');
 readln;
end.
```

```
//Реализация на изброим тип в C++
#include <iostream.h>

// Чрез служебната дума enum на последователност
// от цели числа се задават имената, изброени в{}.
// В случая, sun=0,mon=1 и т.н..
// Има възможност да се укаже началото на изброяването.
// Ако в {} се запише sun=1, останалите имена ще получат
// номера 2,3 и т.н..
// Името Days е име на така създадения тип.
enum Days {sun,mon,tue,wed,thi,fri,sat};
// При извеждане на стойности на променливи от тип Days
// ще се отпечатат съответните числа: 0, 1 ...
// Ако се желае отпечатване на имената, последните трябва
// да бъдат описани чрез символни низове.
// Това е направено чрез следващата декларация:
const char* DNames[7]=
{"Sun","Mon","Tue","Wed","Thu","Fri","Sat"};
// Характерните за C/C++ операции "инкремент"/"декремент"
// не се предварително дефинирани за enum.
// Следва предефиниране на префиксни ++ и --.
Days& operator++(Days &aDay)
{aDay=Days(int(aDay)+1);return aDay;}
Days& operator--(Days &aDay)
{aDay=Days(int(aDay)-1);return aDay;}
//Програмата демонстрира разпечатване на номерата и имената
//на дните от седмицата в права и обратна последователност.
//Използуван е цикъл for с управляваща променлива от тип Days.
void main()
{Days day;
 cout<<"Weekdays enumeration:"<<endl;
 for (day=sun;day<=sat;++day)
     cout<<day<<' '<<DNames[day]<<endl;
// След края на цикъла променливата day има стойност
// ИЗВЪН ДЕФИНИЦИОННАТА ОБЛАСТ!!!
 cout<<day<<endl;
 cout<<"Weekdays countdown:"<<endl;
 for (day=sat;day>=sun;--day)
     cout<<day<<' '<<DNames[day]<<endl;
// След края на цикъла променливата day има стойност
// ИЗВЪН ДЕФИНИЦИОННАТА ОБЛАСТ!!!
 cout<<day<<endl;
}
```

2.7 Разработване на алгоритъм по подхода “отгоре-надолу”

Предлага се задача, подобна на зададената в основното изложение.

```
//Задача: Да се определи в кои квадранти на
//равнинна декартова координатна система лежи
//кръг със зададени координати на центъра и радиус
//Задачата да се реши с минимален брой проверки.
program circles;
{$APPTYPE CONSOLE}
uses  SysUtils;
var x,y,r:real;
begin
repeat   // безкраен цикъл, за да се решават множество случаи
 write('Enter x-y coordinates:');
 readln(x,y);
```

```
 repeat  // цикъл до въвеждане на положителен радиус
  write('Enter radius (>0):');
  readln(r);
 until r>0;
// Решение по подхода 'отгоре-надолу':
// всяка проверка разграничава група случаи;
// всяка група се детайлизира с вложени проверки.

//Координатното начало е вътре в кръга
 if x*x+y*y<r*r then     writeln('1,2,3 and 4')
 else

//Прости случаи: не се пресича ординатната ос
 //Първи и/или четвърти квадрант
 if x>r then
 begin
  if y>r then       writeln('1')
  else if -y>r then writeln('4')
  else              writeln('1 and 4')
 end else
 //Втори и/или трети квадрант
 if -x>r then
 begin
  if y>r then       writeln('2')
  else if -y>r then writeln('3')
  else              writeln('2 and 3')
 end else

//С пресичане на ординатната ос
 begin
  //Прости случаи: два квадранта
  if y>r then       writeln('1 and 2')
  else if -y>r then writeln('3 and 4')
  else

  //С пресичане на две оси
       if x>0 then //1 и 4 с 2 или 3
       begin
        if y>0 then writeln('1,2 and 4')
        else        writeln('1,3 and 4')
       end
       else  //2 и 3 с 1 или 4
       begin
        if y>0 then writeln('1,2 and 3')
        else        writeln('2,3 and 4')
       end;
 end;
until false; //този цикъл се прекъсва с прекъсване на програмата
end.



#include <iostream.h>
//Задача: Да се определи в кои квадранти на
//равнинна декартова координатна система лежи
//кръг със зададени координати на центъра и радиус
//Задачата да се реши с минимален брой проверки.
void main()
{double x,y,r;
for(;;)
{
 cout<<"Enter x-y coordinates and radius:";
 cin>>x>>y>>r;
// Решение по подхода "отгоре-надолу":
```

```
// всяка проверка разграничава група случаи;
// всяка група се детайлизира с вложени проверки.
//Координатното начало е вътре в кръга
 if(x*x+y*y<r*r)cout<<"1,2,3 and 4"<<endl;
 else
//Прости случаи: не се пресича ординатната ос
 //Първи и/или четвърти квадрант
 if(x>r)
 {if(y>r)       cout<<"1"<<endl;
   else if(-y>r)cout<<"4"<<endl;
     else       cout<<"1 and 4"<<endl;
 }else
 //Втори и/или трети квадрант
 if(-x>r)
 {if(y>r)       cout<<"2"<<endl;
   else if(-y>r)cout<<"3"<<endl;
      else      cout<<"2 and 3"<<endl;
 }else
//С пресичане на ординатната ос
 {
  //Прости случаи: два квадранта
  if(y>r)       cout<<"1 and 2"<<endl;
   else if(-y>r)cout<<"3 and 4"<<endl;
    else
  //С пресичане на две оси
     if(x>0)    //1 и 4 с 2 или 3
     {
      if(y>0)   cout<<"1,2 and 4"<<endl;
      else      cout<<"1,3 and 4"<<endl;
     }
     else       //2 и 3 с 1 или 4
     {
      if(y>0)   cout<<"1,2 and 3"<<endl;
      else      cout<<"2,3 and 4"<<endl;
      }
 }
}
}
```

2.8 Процедури

Примерът е върху вече разглеждан алгоритъм, този път кодиран като процедура. Обърнете внимание на разликата в синтаксиса при предаване на параметри по стойност и по адрес.

```
// Задача: Да се пресметне и разпечата броят стъпки, с които
// алгоритъмът на Евклид определя най-големия общ делител
// на зададено цяло число и всички по-малки от него числа.
program EuclProc;
{$APPTYPE CONSOLE}
uses  SysUtils;

// Процедура за пресмятане на най-голям общ делител (GCD).
// Резултатът остава в параметъра a, броят стъпки - в count.
// Затова на процедурата се предават адресите на променливи
// от извикващата програма. Този начин на обмен на данни се
// задава, като пред имената на формалните параметри в
// заглавието на процедурата се поставя служебната дума VAR.
// Процедурата ще работи с чужди (на викащата програма) променливи
// по начина, описан в тялото на процедурата относно променливите
//  a и count.
```

```
procedure Euclid(var a:integer; b:integer; var count:integer);
var r:integer;
begin
 count:=0;           //нулиране на брояча
// Алгоритъм на Евклид, реализиран чрез цикъл със следусловие.
 repeat
  r:=a mod b;
  a:=b;b:=r;
  count:=count+1;   //увеличаване на брояча
 until r=0;
end;

var a1,         //задавано число
     a,         //делимо
     b,         //делител
     c:integer; //брояч
begin
 writeln('Count of Euclid''s operations finding GCD');
 writeln('Enter the dividend number:');
 // Въвеждане на задаваното число със защита от
 // недопустими стойности. В цикъл:
 repeat
  write('An  integer (>2) expected:');  //се извежда подсещане,
  {$I-}readln(a);{$I+}                  //въвежда се число, като временно е
  // подтиснато прекъсването от входно-изходна грешка при въвеждане на
  // недопустими символи (например букви). Цикълът приключва при липса на
 until (IOResult=0)and (a>2);   //грешка на входа и допустима стойност ( >2).
 a1:=a;//Процедурата променя a, затова въведената стойност се съхранява в a1.
//Цикъл по всички възможни делители
 for b:=2 to a1-1 do
 begin
  a:=a1;//Задаване на началната стойност на делимото
  Euclid(a,b,c);//Пресмятане на GCD и броя стъпки
  // Извеждане на резултата
  writeln('GCD of ',a1,' and ',b,' is ',a,
        ' ,found at ',c,' iteration.');
 end;// Край на цикъла
 writeln('Press <Enter> to close');readln;
end.



// Задача: Да се пресметне и разпечата броят стъпки, с които
// алгоритъмът на Евклид определя най-големият общ делител
// на зададено цяло число и всички по-малки от него числа.
#include <iostream.h>
#include <cstdlib>
// Процедура за пресмятане на най-голям общ делител (GCD).
// Резултатът остава в параметъра a, броят стъпки - в count.
// Затова на процедурата се предават адресите на променливи
// от извикващата програма. Този начин на обмен на данни се
// задава, като пред имената на формалните параметри в
// заглавието на процедурата се поставя знакът &. В C++ е
// прието такива параметри да се наричат параметри-псевдоними.
// Този термин е оправдан от факта, че употребените символични
// имена (в случая - a и count) са локални за процедурата имена
// на чужди (на викащата програма) променливи.
void Euclid(int &a,int b, int &count)
{int r;
 count=0;     //нулиране на брояча
// Алгоритъм на Евклид, реализиран чрез цикъл със следусловие.
 do
 {r=a%b;
  a=b;b=r;
  count+=1;   //увеличаване на брояча
```

```
 }while(r!=0);
}

void main()
{int a1,       //задавано число
      a,       //делимо
      b,       //делител
      c;       //брояч
 char buf[10]; //буфер за въвеждане

 cout<<"Count of Euclid's operations finding GCD\n";
 cout<<"Enter the dividend number:\n";
 // Въвеждане на задаваното число със защита от
 // недопустими стойности. В цикъл:
 do
 {cout<<"An  integer (>2) expected:";  //се извежда подсещане,
  cin.getline(buf,9);                  //въвеждат се символи
                                       //от клавиатурата
  a=atoi(buf);                         //и се преобразуват в число,
 }while(a<2);                          //което трябва да е >2.
 a1=a;//Процедурата променя a, затова въведената стойност се запазва
//Цикъл по всички възможни делители
 for (b=2;b<a1;b++)
 {a=a1;//Задаване на началната стойност на делимото
  Euclid(a,b,c);//Пресмятане на GCD и броя стъпки
  // Извеждане на резултата
  cout<<"GCD of "<<a<<" and "<<b<<" is "<<a
        <<" ,found at "<<c<<" iteration.\n";
 }// Край на цикъла
}
```

2.9 Функции

Предлаганият прост пример решава задачата за извеждане на целите числа от зададен интервал, които се делят на зададено число и същевременно не се делят на друго зададено число.

```
program divisibility;
{$APPTYPE CONSOLE}
uses SysUtils;
function divide(a,b:integer):boolean;
begin divide:=a mod b =0; end;

var a,b,c,d,x:integer;
begin
 writeln('Report all integers between A and B, ',
       'divisible by C but not divisible by D. ');
 writeln('Enter limits (A<B): ');
 write ('A: ');readln(a);
 write ('B: ');readln(b);
 writeln('Enter the two dividers: ');readln(c,d);
 writeln('Selected numbers: ');
 for x:=a to b do
  if divide(x,c)and not divide(x,d) then write(x,' ');
 writeln;readln;
end.
```

```
#include <iostream.h>

bool divide(int a,int b)
{return a%b==0;}

void main()
{int a,b,c,d;
 cout<<"Report all integers between A and B, "
       <<"divisible by C but not divisible by D."<<endl;
 cout<<"Enter limits (A<B):"<<endl;
 cout<<"A: ";cin>>a;
 cout<<"B: ";cin>>b;
 cout<<"Enter the two dividers: ";cin>>c>>d;
 cout<<"Selected numbers:"<<endl;
 for(int x=a;x<=b;x++)
       if(divide(x,c)&&!divide(x,d))cout<<x<<' ';
 cout<<endl;
}
```

Следващият програмен текст илюстрира работа със символни и низови променливи, като предлага решение на задачата за представяне и събиране на числа в позиционна бройна система с основа от 2 до 16.

```
//Представяне и събиране на числа в произволна позиционна бройна система.
program nbases;
{$APPTYPE CONSOLE}
uses SysUtils;
//-----------------------------ПОМОЩНИ ФУНКЦИИ-----------------------------
//Функция за прекодиране на символен низ от кирилица Win към конзолна
function cnv(s:string):string;
var i:BYTE;c:CHAR;
begin
for i:=1 to length(s) do
begin c:=s[i];
if (c>='А')and(c<='я') then
//Премахнете следващия ред, ако нямате разкъсано разположение
if c>'п' then s[i]:=char(byte(c)-16) else
//на малките букви в конзолната кирилица
s[i]:=char(byte(c)-64);
end;
cnv:=s;
end;
//_____________________________________________________________________
//Преобразуване на малка буква от латиницата в главна
function UpCL( c:char):char;
begin
if (c>='a') and (c<='z') then UpCL:=char(byte(c)-32)
else UpCL:=c;
end;
//_____________________________________________________________________
//Преобразуване на цифра (знак) в число.
//При бройни системи с основа n>10 за цифри се вземат латинските букви.
function chtn(c:char):integer;
begin
c:=UpCL(c);            //За всеки случай, ако са използвани малки букви
if (c>='0')and(c<='9')then chtn:= integer(c)-integer('0')
else if(c>='A')and(c<='F')then chtn:=10 + byte(c)-byte('A')
else begin writeln('Illegal symbol ',c);chtn:=-1;end;
end;//Връща се съответното на цифрата цяло число или -1, ако цифрата е лоша.
```

```
//_____________________________________________________________________________
//Връща подадения символен низ, нареден в обратен ред
function invstr(var s:string):string;
var c:CHAR;i,n:BYTE;
begin n:= length(s);
for i:=1 to n div 2 do
begin c:=s[i];s[i]:=s[n+1-i];s[n+1-i]:=c;end;
invstr:=s;     //s остава инвертиран (обърнат)
end;
//-------------------ПРЕДСТАВЯНЕ НА ЧИСЛО ПРИ ДРУГА ОСНОВА-------------------
//_____________________________________________________________________________
//Създава символен низ, представящ числото a в поз.бр.система с основа n.
function nstr(a:longint;n:BYTE):string;
const dig:string[16]='0123456789ABCDEF';
var s:string;d:BYTE;
begin
s:=''; while a<>0 do
begin d:=a mod n;a:=a div n;s:=dig[d+1]+s;end;
if s='' then nstr:='0' else nstr:=s;
end;
//Тест за представяне на десетично число в друга бройна система
procedure NStrTest;
var num:longint;base:BYTE;
const prompt:string='Input a base N (N>=2,N<=16,otherwise stop) and a number:';
begin
repeat //ще повтаря, докато не се въведе основа, различна от посоченото
write(prompt);readln(base);
if(base<2)or(base>16)then break;readln(num);
writeln('The number ',num,' based on ',base,' is ',nstr(num,base));
until false;
end;
//------------ПРЕСМЯТАНЕ НА ПРЕДСТАВЕНО ПРИ ДРУГА ОСНОВА ЧИСЛО-----------
//_____________________________________________________________________________
//Символният низ s съдържа цифрите на число, представено в n-ична бр.с.
//Функциятя връща стойността на числото.
function strn(s:string;n:BYTE):longword;
var v:longword;i:BYTE;t:integer;const INT_MAX=$FFFFFFFF;
begin
v:=0;
for i:=1 to length(s) do
begin
//Следва проверка за очаквано препълване на v
if v>INT_MAX div n then begin Writeln('Overflow!');strn:=INT_MAX;exit;end;
t:=chtn(s[i]);                    //Взема се стойността на текущата цифра
//и ако всичко е наред, v се умножава по основата и се прибавя t
if(t>=0)and(t<n)then begin v:=v*n;v:=v+t;end
else begin writeln('Calculation aborted. Current value',v);break;end;
end;
strn:=v;
end;
//Тест за пресмятане на число, записано в друга бройна система
procedure StrnTest;
var s:string;n:BYTE;
begin
repeat
write('Input the base (>=2,<=16 or other to stop):');Readln(n);
if(n<2)or(n>16)then break;
write('Input a string for the number:');readln(s);
writeln('Decimal representation for: ',s,' is ',strn(s,n));
until false;
end;
//-----------------------СЪБИРАНЕ ПРИ ДРУГА ОСНОВА--------------------------
//_____________________________________________________________________________
//Символните низове a и b съдържат числа в позиционна бройна система
```

```
//с основа n (n<=16). Функцията връща символен низ – тяхната сума.
function sum(a,b:string;n:BYTE):string;
const dig:string[16]='0123456789ABCDEF';
var p,i,v:BYTE;a2,b2:integer;s:string;
begin
//Сумирането започва отдясно (цифрите на единиците), а отляво по-късият низ
//се допълва с нули.Затова низовете се инвертират и се допълва отдясно.
a:=invstr(a); b:=invstr(b);
a2:=length(a);b2:=length(b);//намира се дължината на двата низа
if a2<b2 then v:=b2 else v:=a2;     //v е по-голямата от двете дължини
for i:=a2+1 to v do a:=a+'0';       //допълване с нули
for i:=b2+1 to v do b:=b+'0';       //не се знае кой низ е по-късият
//От тук започва самият алгоритъм за събиране
s:='';p:=0;
for i:=1 to length(a) do
begin
a2:=chtn(a[i]);b2:=chtn(b[i]);  //поредните цифри се преобразуват в числа
if (a2<0)or(b2<0)or(a2>=n)or(b2>=n)then        //ако има недопустима цифра
begin sum:='Wrong symbol.Sum aborted.';exit;end; //прекъсва събирането
v:=a2+b2+p;p:=v div n;v:=v mod n;  //пресмята се поредна цифра v и пренос p
s:=dig[v+1]+s;          //отляво в низа-резултат се записва поредната цифра
end;
if p>0 then s:=dig[p+1]+s;//След края на цикъла се добавя цифрата от преноса
sum:=s;
end;
//Тест за събиране в произволна позиционна бройна система
procedure SumTest;
const mess:string='Събиране в произволна позиционна бройна система.';
prompt:string=
'Въведете поотделно основа (<=16) и два символни низа – събираемите.';
var n:BYTE;s1,s2:string;
begin
repeat       //ще повтаря, докато не се въведе основа, различна от посоченото
writeln(cnv(mess));writeln(cnv(prompt));
readln(n);if(n<2)or(n>16)then break;
readln(s1);readln(s2);
write('The sum of ',s1,' and ',s2,' is ');
writeln(sum(s1,s2,n));
until false;
end;
//_____________________________________________________________________________
const mess:string='ПРЕДСТАВЯНЕ НА ЧИСЛА В ПОЗИЦИОННА БРОЙНА СИСТЕМА';
begin
writeln(cnv(mess));NStrTest;
writeln('Converting numbers from other bases to decimal.');StrnTest;
SumTest;
Write('Press <Enter> to close...');Readln;
end.
```

```
#include <iostream.h>
#include <limits.h> //за да извлечем максималното в средата цяло число
typedef unsigned int WORD;//за работа с цяло без знак- WORD е по-къса дума
//----------------------------ПОМОЩНИ ФУНКЦИИ----------------------------
//Функция за прекодиране на символен низ от кирилица Win към конзолна
char * cnv(char *s)
{for (int i=0; s[i];i++)
{char c; c=s[i];
if ((c>='A') && (c<='я'))
//Премахнете следващия ред, ако нямате разкъсано разположение
//на малките букви в конзолната кирилица
if (c>'п') s[i]=char(int(c)-16);else
s[i]=char(int(c)-64);
```

```
}
return s;                  //Връща се преобразуваният изходен символен низ.
}
//_____________________________________________________________________________
//Преобразуване на малка буква от латиницата в главна
char UpCL(char c)
{if ((c>='a') && (c<='z')) return char(int(c)-32);
else return c;
}
//_____________________________________________________________________________
//Преобразуване на цифра (знак) в число.
//При бройни системи с основа n>10 за цифри се вземат латинските букви.
int chtn(char c)
{c=UpCL(c);                //За всеки случай, ако са използвани малки букви
if ((c>='0')&&(c<='9')) return int(c)-int('0');
else if((c>='A')&&(c<='F')) return 10 + int(c)-int('A');
else {cout<<"Illegal symbol "<<c<<endl; return -1;}
}   //Връща се съответното на цифрата цяло число или -1, ако цифрата е лоша.
//_____________________________________________________________________________
//Връща подадения символен низ, нареден в обратен ред
char * invstr(char * s)
{char c;int n=0;
while(s[n])n++;                 //Равносилно е на n=strlen(s)
for (int i=0;i<n/2;i++){c=s[i];s[i]=s[n-1-i];s[n-1-i]=c;}
return s;                       //s остава инвертиран (обърнат)
}
//-------------------ПРЕДСТАВЯНЕ НА ЧИСЛО ПРИ ДРУГА ОСНОВА-------------------
//_____________________________________________________________________________
//Създава символен низ, представящ числото a в поз.бр.система с основа n.
char * nstr (int a, WORD n)
{static char s[80], dig[]="0123456789ABCDEF";
WORD i=0,d;s[0]='0';
while (a!=0){d=a%n;a=a/n;s[i++]=dig[d];}
if (i) s[i]='\0';else s[1]='\0';
return invstr(s);
}
//Тест за представяне на десетично число в друга бройна система
void NStrTest()
{char prompt[]="Input a base N (N>=2,N<=16,otherwise stop) and a number:";
int num,base;
do{       //ще повтаря, докато не се въведе основа, различна от посоченото
cout<<prompt; cin>>base>>num;
if(base<2||base>16)break;
cout<<"The number"<<num<<" based on "<<base<<" is"<<nstr(num,base))<<endl;
}while(base<=16);
}
//------------ПРЕСМЯТАНЕ НА ПРЕДСТАВЕНО ПРИ ДРУГА ОСНОВА ЧИСЛО-----------
//_____________________________________________________________________________
//Символният низ s съдържа цифрите на число, представено в n-ична бр.с.
//Функциятя връща стойността на числото.
WORD strn(char * s,WORD n)
{WORD v=0,i=0,t;        //Стойността се набира в v, i е брояч на позицията
while (s[i])             //докато има цифри
//Следва проверка за очаквано препълване на v
{if(v>UINT_MAX/n){cout<<"Overflow!\n";return UINT_MAX;};
t=chtn(s[i]);                          //Взема се стойността на текущата цифра
//и ако всичко е наред, v се умножава по основата и се прибавя t
if((t>=0)&&(t<n)){v*=n;v+=t;i++;}
else {cout<<"Calculation aborted. Current value "<<v<<endl;break;};
}return v;
}
//Тест за пресмятане на число, записано в друга бройна система
void StrnTest()
{char s[20]; int n;
```

```
do{
cout<<
"Input the base (>=2,<=16 or other to stop) and a string for the number:";
cin>>n>>s);if(n<2||n>16)break;
cout<<"Decimal representation for: "<<s <<" is "<< strn(s,n))<<endl;
}while (n<=16);
}
//------------------------СЪБИРАНЕ ПРИ ДРУГА ОСНОВА---------------------------
//_____________________________________________________________________________
//Символните низове a и b съдържат числа в позиционна бройна система
//с основа n (n<=16). Функцията връща символен низ - тяхната сума.
char * sum (char * a,char * b,WORD n)
{static char s[80],dig[]="0123456789ABCDEF";
int p=0,i,v,a2,b2;
//Сумирането започва от цифрите на единиците, а отляво по-късият низ
//се допълва с нули.Затова низовете се инвертират, а допълването става
//отдясно. След сумиране резултатът се инвертира обратно.
a=invstr(a); b=invstr(b);
i=0;while (a[i])i++;a2=i;               //намира се дължината на двата низа
i=0;while (b[i])i++;b2=i;
v=(a2>b2?a2:b2);                        //v е по-голямата от двете дължини
for(i=a2;i<v;i++)a[i]='0';a[i]='\0';     //допълване с нули
for(i=b2;i<v;i++)b[i]='0';b[i]='\0';     //не се знае кой низ е по-късият
//От тук започва самият алгоритъм за събиране
for (i=0;a[i];i++)                      //докато има цифри
{a2=chtn(a[i]);b2=chtn(b[i]);     //поредните цифри се преобразуват в числа
if (a2<0||b2<0||a2>=n||b2>=n)     //ако има недопустима цифра
return"Wrong representation. Sum aborted.\n";  //прекъсва събирането
v=a2+b2+p;p=v/n;v=v%n;           //пресмятат се поредната цифра v и преносът p
s[i]=dig[v];                     //записва се поредната цифра в низа - резултат
}                              //След края на цикъла се добавя цифрата от преноса
if(p>0)s[i++]=dig[p];s[i]='\0';   //и се затваря низът с \0
return invstr(s);                 //връща се инвертираният низ
}
//Тест за събиране в произволна позиционна бройна система
void SumTest()
{int n; char s1[10],s2[10],*m,*p;
char mess[]="Събиране в произволна позиционна бройна система.\n";
char prompt[]="Въведете основа (<=16) и два символни низа - събираемите.\n";
m=cnv(mess);p=cnv(prompt);
do{            //ще повтаря, докато не се въведе основа, различна от посоченото
cout<<m<<' '<<p);
cin>>n>>s1<<s2; if(n<2||n>16)break;
cout<<"The sum of "<< s1<<" and "<<s2<<" is ";
cout<<sum(s1,s2,n))<<endl;
}while (n<=16);
}
//_____________________________________________________________________________
int main(int argc, char* argv[])
{char mess[]="ПРЕДСТАВЯНЕ НА ЧИСЛА В ПОЗИЦИОННА БРОЙНА СИСТЕМА";
cout<<cnv(mess))<<endl;
NStrTest();
cout<<"Converting numbers from other bases to decimal.\n";
StrnTest();
SumTest();
return 0;
}
```

2.10 Съставни типове данни

Приложените текстове демонстрират използването на масиви за работа с числови вектори и матрици.

```
program Matrices;
{$APPTYPE CONSOLE}
uses SysUtils;
const Matsize=50;
type Row=array [1..MatSize] of real;   //Ред от матрица
     Matrix=array [1..MatSize] of Row; //Вектор от редове = матрица

//Транспониране на квадратна матрица
procedure TransMatr2(var A:Matrix; N:Byte);
 var i,j:Byte; c:Real;
 // За да се избегне двукратната размяна на елементи при обхождане на цялата
 // матрица, обхождат се само елементите под главния диагонал  и се разменят
 // със симетричните им
 begin
  for i:=2 to N do  for j:=1 to i-1 do
  begin c:=A[i,j]; A[i,j]:=A[j,i]; A[j,i]:=c; end;
 end;

// Транспониране на правоъгълна матрица
procedure Transp(var A:Matrix; N,M:BYTE);
var i,j,K:BYTE;
begin
 // Първо се транспонира квадратната подматрица
 if N<M then K:=N else K:=M;
 TransMatr2(A,K);
 // После се пренася отдясно оставащата отдолу част
 if K<N then
  for i:=K+1 to N do for j:=1 to M do A[j,i]:=A[i,j]
 // или отдолу оставащата отдясно част
 else
  for j:=K+1 to M do for i:=1 to N do A[j,i]:=A[i,j];
end;

// Транспониране на  матрица оформено като функция.
// В стандартен Pascal това не е възможно
// Служебната дума const указва, че параметърът се предава по адрес,
// но без право да бъде модифициран.
function Trans(const A:Matrix; N,M:BYTE):Matrix;
var i,j:BYTE;
 begin
  for i:=1 to N do for j:=1 to M do Trans[j,i]:=A[i,j];
 end;

// Умножение на две матрици A и B с размери  N x M и  M x L .
//Резултатът е матрицата C с размер  N x L .
procedure MatrixProduct(const A,B:Matrix;var C:Matrix; N,M,L:BYTE);
var i,j,k:BYTE; t:real;
begin
 for i:=1 to N do for j:=1 to L do
 begin t:=0; //По-ефективно е сумата да се натрупва в проста променлива
  for k:=1 to M do t:=t+A[i,k]*B[k,j];
  C[i,j]:=t; //и после да се присвоява на елемент от резултата
 end;
end;


// Скаларно произведение на вектори.
// Служи за умножение на ред по стълб на умножаваните матрици.
```

```
function ScalPro(const A,B:Row; N:BYTE):real;
var k:BYTE; t:real;
begin
  t:=0;for k:= 1 to N do t:=t+A[k]*B[k];
  ScalPro:=t;
end;

//Връща вектор-стълб с номер number от матрица A с  N реда
function Column(const A:Matrix; N,number:BYTE):Row;
var k:BYTE;
begin
 for k:=1 to N do Column[k]:=A[k,number];
end;

//Умножение на две матрици A с размери  M x N и B с размери  N x K .
//Резултатът е матрицата C  с размери  M x K .
procedure MatrProdBySP(const A,B:Matrix;var C:Matrix;M,N,K:BYTE);
var i,j:BYTE;
begin
  for i:=1 to M do for j:=1 to K do
  C[i,j]:=ScalPro(A[i],Column(B,N,j),N);
  //A[i] е i-тият ред от матрицата A . Column(B,N,j) дава j-тия стълб от B.
  //N е общата им дължина.
end;


//Извежда вектор. Може да се използува и за извеждане на ред от матрица.
procedure PrintRow(const A:Row; M:BYTE);
var j,k,na:BYTE;
begin
  na:=1;
  //Следващият цикъл извежда по 10 числа на печатен ред до изчерпване на реда
   repeat
    k:=na+9; if k>M then k:=M;
    for j:=na to k-1 do Write(A[j]:8:4);Writeln(A[k]:8:4);
    na:=k+1;
   until na>M;
end;

// Извеждане матрица A с размер  N*M
procedure PrintMatrix(const A:Matrix; N,M:Byte);
 var i:BYTE;
 begin
  Writeln('Printing matrix ', N:2, ' by ', M:2);
  //Цикъл по редове
  for i:= 1 to N do
  begin
   Writeln('           Row ',i:3,' :');
   PrintRow(A[i],M);
  end;//cycle
 end; //PrintMatrix

// Генериране на правоъгълна матрица от случайни числа
procedure GenerateMatrix(var A:Matrix; N,M:Byte);
var i,j:BYTE;
begin
  for i:=1 to N do
   for j:=1 to M do
    A[i,j]:=(random(1000)-500)/500.0;
end;

var A,B,C:Matrix;
    N,M:BYTE;
begin
```

```
Writeln('Matrices operations Demo.',#13,#10,'Input Matrix Sizes:');
Readln(N, M);
Randomize;
// Генерира и извежда правоъгълна матрица
GenerateMatrix(A,N,M);PrintMatrix(A,N,M);
write('Press <Enter> to continue after printing.');Readln;
// Транспонира я и извежда транспонираната
writeln('Transposed matrix:');
Transp(A,N,M);PrintMatrix(A,M,N);Readln;
// Чрез повторно транспониране се възстановява
writeln('Transposition of the transposed matrix:');
Transp(A,M,N);PrintMatrix(A,N,M);Readln;
// Подготвя се втора матрица
writeln('A new matrix:');
GenerateMatrix(B,M,N);PrintMatrix(B,M,N);Readln;
// Прави се тяхното произведение
writeln('The product of last two matrices:');
MatrixProduct(A,B,C,N,M,N);PrintMatrix(C,N,N);Readln;
// Проверка чрез транспонирания резултат на
// комутираното произведение на транспонираните матрици
writeln('The same product using transposed matrices:');
MatrProdBySP(Trans(B,M,N),Trans(A,N,M),C,N,M,N);
PrintMatrix(Trans(C,N,N),N,N);Readln;
end.
```

```
#include <iostream.h>
#include <cstdlib>
const int MatSize=50;
typedef float Row[MatSize];   //Ред от матрица
typedef Row Matrix[MatSize];  //Вектор от редове = матрица
//Транспониране на квадратна матрица
void TransMatr2(Matrix A,int N)
{int i,j;float c;
 // За да се избегне двукратната размяна на елементи при обхождане на цялата
 // матрица, обхождат се само елементите под главния диагонал  и се разменят
 // със симетричните им
  for (i=1;i< N;i++)for (j=0;j<=i-1;j++)
  {c=A[i][j]; A[i][j]=A[j][i]; A[j][i]=c;}
}

// Транспониране на правоъгълна матрица
void Transp(Matrix A,int N,int M)
{int i,j,K;
 // Първо се транспонира квадратната подматрица
 if (N<M)K=N; else K=M;
 TransMatr2(A,K);
 // После се пренася отдясно оставащата отдолу част
 if (K<N)
  for (i=K;i<N;i++)for (j=0;j<M;j++)A[j][i]=A[i][j];
 // или отдолу оставащата отдясно част
 else
  for (j=K;j<M;j++)for (i=0;i<N;i++)A[j][i]=A[i][j];
}

// Умножение на две матрици A и B с размери  N x M и  M x L .
//Резултатът е матрицата C с размер  N x L .
void MatProd(const Matrix A,const Matrix B,Matrix C,int N,int M,int L)
{int i,j,k;float t;
 for(i=0;i<N;i++)for(j=0;j<L;j++)
 {t=0; //По-ефективно е сумата да се натрупва в проста променлива
  for(k=0;k<M;k++)t+=A[i][k]*B[k][j];
  C[i][j]=t; //и после да се присвоява на елемент от резултата
 }
```

```
}

// Вмъкване на форматиращи символи в изходния поток данни
ostream& setfld(ostream &str)
{str.width(7);str.precision(3);return str;}

//Извежда вектор. Може да се използува и за извеждане на ред от матрица.
void PrintRow(const Row A,int M)
{int j,k,na=0;
  //Следващият цикъл извежда по 10 числа на печатен ред до изчерпване на реда
   do
   {k=na+10; if(k>=M-1) k=M-1;
    for(j=na;j<k;j++)cout<<setfld<<A[j];
      cout<<setfld<<A[j]<<endl;
    na=k+1;
   }while(na<M);
}

// Извеждане матрица A с размер  N*M
void PrintMatrix(const Matrix A,int N,int M)
{int i;
 cout<<"Printing matrix "<<N<<" by "<<M<<endl;
  //Цикъл по редове
  for(i=0;i<N;i++)
  {cout<<"           Row "<<i<<" :"<<endl;
   PrintRow(A[i],M);
  }
}

// Генериране на правоъгълна матрица от случайни числа
void GenerateMatrix(Matrix A,int N,int M)
{int i,j;
 for(i=0;i<N;i++)
   for(j=0;j<M;j++)
    A[i][j]=0.002*(rand()%1000-500);
}
void main()
{Matrix A,B,C;
 int N,M;
cout<<"Matrices operations Demo. \n Input Matrix Sizes:";
cin>>N>>M;
// Генерира и извежда правоъгълна матрица
GenerateMatrix(A,N,M);PrintMatrix(A,N,M);
// Транспонира я и извежда транспонираната
cout<<"Transposed matrix:"<<endl;
Transp(A,N,M);PrintMatrix(A,M,N);
// Чрез повторно транспониране се възстановява
cout<<"Transposition of the transposed matrix:"<<endl;
Transp(A,M,N);PrintMatrix(A,N,M);
// Подготвя се втора матрица
cout<<"A new matrix:"<<endl;
GenerateMatrix(B,M,N);PrintMatrix(B,M,N);
// Прави се тяхното произведение
cout<<"The product of last two matrices:"<<endl;
MatProd(A,B,C,N,M,N);PrintMatrix(C,N,N);
// Проверка чрез транспонирания резултат на
// комутираното произведение на транспонираните матрици
cout<<"The same product using transposed matrices:"<<endl;
Transp(B,M,N);Transp(A,N,M);
MatProd(B,A,C,N,M,N);Transp(C,N,N);PrintMatrix(C,N,N);
}
```

2.11 Сортиране

Предлагат се процедури, реализиращи така наречените “примитивни” методи за сортиране на едномерен масив: метод на пряката селекция, метод на прякото вмъкване, метод на мехурчето. Дадени са и две разновидности на метода на мехурчето: със смяна на посоката на обхождане и с променлива стъпка на сравнения и размяна. Методът на мехурчето с променлива стъпка, предложен през 1991г. се оказва изненадващо по- производителен от другите примитивни методи. Чрез директивата {$DEFINE TEST}, респ. #define test се включва компилация и изпълнение на добавени в текста оператори, отброяващи проверките и разместванията, употребени от всяка от процедурите при сортиране на един и същи масив от случайни числа. Това отброяване може да се изключи с премахване на знака за коментар пред директивата UNDEF.

```
program sorts;

{$APPTYPE CONSOLE}
{$DEFINE TEST}
//{$UNDEF TEST}
uses  SysUtils;

const sz=10000;
var  CNTP,CNTA:integer;
type T=integer;
     TAr=array[0..sz-1]of T;

// Метод на пряката селекция
procedure SelSort(var A:TAr; N:integer);
var i,j,imin:integer;
    min:T;
begin
{$ifdef TEST}
  CNTP:=0;CNTA:=0;
{$endif}
 for i:=0 to N-2 do
 begin
  imin:=i;min:=A[imin];
  for j:=i+1 to N-1 do
  begin
   if A[j]<min then
   begin min:=A[j];imin:=j;end;
   {$ifdef TEST}CNTP:=CNTP+1;{$endif}
  end;
  A[imin]:=A[i];A[i]:=min;
  {$ifdef TEST}CNTA:=CNTA+1;{$endif}
 end;
end;

// Генериране на масив от случайни числа
procedure GenArr(var A:TAr;N:integer);
var i:integer;
begin for i:=0 to N-1 do A[i]:=random(1000);end;
// Копиране на масив
procedure CpyArr(var A,B:TAr;N:integer);
var i:integer;
begin for i:=0 to N-1 do A[i]:=B[i];end;
// Печатане на масив
procedure PrnArr(var A:TAr;N:integer);
```

```
var i:integer;
begin
 for i:=0 to N-1 do
  if (i+1) mod 10 =0 then writeln(A[i]:7)else write(A[i]:7);
  writeln;
end;

// Метод на прякото вмъкване
procedure InsSort(var A:TAr ; N:integer);
var i,j:integer;t1:T;
begin
{$ifdef TEST}CNTP:=0;CNTA:=0;{$endif}
 for i:=1 to N-1 do
 begin t1:=A[i];j:=i-1;
 while(j>=0) and (A[j]>t1) do
 begin
 {$ifdef TEST}inc(CNTP);inc(CNTA);{$endif}
  A[j+1]:=A[j];dec(j);
 end;
 A[j+1]:=t1;
 end;
end;
// Метод на мехурчето
procedure BblSort(var A:TAr ; N:integer);
var  i,L:integer;inv:boolean;t1:T ;
begin
{$ifdef TEST}CNTP:=0;CNTA:=0;{$endif}
L:=N;inv:=true;
 while inv do
 begin inv:=false;
  for i:=0 to L-2 do
  begin
   {$ifdef TEST}CNTP:=CNTP+1;{$endif}
   if A[i]>A[i+1]then
   begin
    t1:=A[i];A[i]:=A[i+1];A[i+1]:=t1;
    inv:=true;
    {$ifdef TEST}CNTA:=CNTA+1;{$endif}
   end;
  end;
  L:=L-1;
 end;
end;

// Метод на мехурчето с двупосочно обхождане
procedure ShkSort(var A:TAr ; N:integer);
var  i,L,F:integer;inv:boolean;t1:T ;
begin
{$ifdef TEST}CNTP:=0;CNTA:=0;{$endif}
L:=N-1;F:=0;inv:=true;
 while inv do
 begin inv:=false;
  for i:=F to L-1 do
  begin if A[i]>A[i+1] then
  begin t1:=A[i];A[i]:=A[i+1];A[i+1]:=t1;
  {$ifdef TEST}inc(CNTA);{$endif}
  end;
  {$ifdef TEST}inc(CNTP);{$endif}
  end;
  dec(L);
  for i:=L downto F+1 do
  begin if A[i]<A[i-1] then
  begin t1:=A[i];A[i]:=A[i-1];A[i-1]:=t1;
   inv:=true;
```

```
   {$ifdef TEST}inc(CNTA);{$endif}
  end;
  {$ifdef TEST}inc(CNTP);{$endif}
 end;
 inc(F);
 end;//while
end;//ShkSort

// Метод на мехурчето с променлива стъпка
procedure BImSort(var A:TAr ; N:integer);
var  i,S:integer;inv:boolean;t1:T ;
begin
{$ifdef TEST}CNTP:=0;CNTA:=0;{$endif}
S:=N*7 div 10;
inv:=true;
 while inv or (S>1)do
 begin inv:=false;
  for i:=0 to N-S-1 do
  begin
   {$ifdef TEST}CNTP:=CNTP+1;{$endif}
   if A[i]>A[i+S]then
   begin
    t1:=A[i];A[i]:=A[i+S];A[i+S]:=t1;
    inv:=true;
    {$ifdef TEST}CNTA:=CNTA+1;{$endif}
   end;
  end;
  if S>1 then S:=S*7 div 10;
 end;//while
end;

var A,B:TAr; N:integer;
begin
 write('Enter the number of elements:');
 readln(N);
 GenArr(B,N);
 writeln('Selection Sort: ');
 CpyArr(A,B,N);
 writeln('Unsorted array:');
 PrnArr(A,N);
 writeln('Sorted array:');
 SelSort(A,N);
 PrnArr(A,N);
{$ifdef TEST}
 writeln('Checks: ',CNTP,' Swaps: ',CNTA);
 {$endif}
 writeln('Insertion Sort: ');
 CpyArr(A,B,N);
 writeln('Unsorted array:');
 PrnArr(A,N);
 writeln('Sorted array:');
 InsSort(A,N);
 PrnArr(A,N);
 {$ifdef TEST}
 write('Checks: ',CNTP,' Swaps: ',CNTA);writeln;;
 {$endif}
 writeln('Bubble Sort: ');
 CpyArr(A,B,N);
 writeln('Unsorted array:');
 PrnArr(A,N);
 writeln('Sorted array:');
 BblSort(A,N);
 PrnArr(A,N);
 {$ifdef TEST}
```

```
 write('Checks: ',CNTP,' Swaps: ',CNTA);writeln;;
 {$endif}
 writeln('Shake Sort: ');
 CpyArr(A,B,N);
 writeln('Unsorted array:');
 PrnArr(A,N);
 writeln('Sorted array:');
 ShkSort(A,N);
 PrnArr(A,N);
 {$ifdef TEST}
 write('Checks: ',CNTP,' Swaps: ',CNTA);writeln;;
 {$endif}
 writeln('Improved bubble Sort: ');
 CpyArr(A,B,N);
 writeln('Unsorted array:');
 PrnArr(A,N);
 writeln('Sorted array:');
 BImSort(A,N);
 PrnArr(A,N);
 {$ifdef TEST}
 write('Checks: ',CNTP,' Swaps: ',CNTA);writeln;;
 {$endif}
  readln;
end.
```

```
#include <iostream.h>
#include <cstdlib>
#define TEST
//#undef TEST
const int sz=10000;
int CNTP,CNTA;

typedef int T;
typedef T TAr[sz];

// Метод на пряката селекция
void SelSort(TAr A,int N)
{int i,j,imin;
 T min;
#ifdef TEST
  CNTP=CNTA=0;
#endif
 for (i=0;i<N-1;i++)
 {imin=i;min=A[imin];
  for(j=i+1;j<N;j++)
  {if(A[j]<min)
   {min=A[j];imin=j;}
#ifdef TEST
  CNTP++;
#endif
  }
  A[imin]=A[i];A[i]=min;
#ifdef TEST
  CNTA++;
#endif
 }
}

// Метод на мехурчето
void BblSort(TAr A,int N)
{int i,L=N;bool inv=true;T t;
#ifdef TEST
  CNTP=CNTA=0;
```

```
#endif
 while(inv)
 {inv=false;
  for(i=0;i<L-1;i++)
  {
#ifdef TEST
  CNTP++;
#endif
  if(A[i]>A[i+1])
  {t=A[i];A[i]=A[i+1];A[i+1]=t;
   inv=true;
#ifdef TEST
  CNTA++;
#endif
  }
  }
  L--;
 }
}



// Метод на мехурчето с двупосочно обхождане
void ShkSort(TAr A,int N)
{int i,L=N-1,F=0;
 bool inv=true;T t;
#ifdef TEST
  CNTP=CNTA=0;
#endif
 while(inv)
 {inv=false;
  for(i=F;i<L;i++)
  {if(A[i]>A[i+1])
  {t=A[i];A[i]=A[i+1];A[i+1]=t;
#ifdef TEST
  CNTA++;
#endif
  }
#ifdef TEST
  CNTP++;
#endif
  }L--;
  for(i=L;i>F;i--)
  {if(A[i]<A[i-1])
  {t=A[i];A[i]=A[i-1];A[i-1]=t;
   inv=true;
#ifdef TEST
  CNTA++;
#endif
  }
#ifdef TEST
  CNTP++;
#endif
 }F++;
 }
}


// Метод на мехурчето с променлива стъпка
void BImSort(TAr A,int N)
{int i,S=0.8*N;T t;bool inv=true;
#ifdef TEST
  CNTP=CNTA=0;
```

```
#endif
 while(inv||S>1)
 {inv=false;
  for(i=0;i<N-S;i++)
  {
#ifdef TEST
   CNTP++;
#endif
   if(A[i]>A[i+S])
   {t=A[i];A[i]=A[i+S];A[i+S]=t;inv=true;
#ifdef TEST
  CNTA++;
#endif
   }
  }S=S>1?S*0.7:1;
 }
}



// Метод на прякото вмъкване
void InsSort(TAr A,int N)
{int i,j;T t;
#ifdef TEST
  CNTP=CNTA=0;
#endif
 for(i=1;i<N;i++)
 {t=A[i];j=i-1;
 while(j>=0&&A[j]>t)
 {
#ifdef TEST
  CNTP++;CNTA++;
#endif
  A[j+1]=A[j--];
 }
 A[j+1]=t;
 }
}

// Генериране на масив от случайни числа
void GenArr(int A[],int N)
{for (int i=0;i<N;i++)A[i]=rand()%1000;}

// Копиране на масив
void CpyArr(int A[],int B[],int N)
{for (int i=0;i<N;i++)A[i]=B[i];}

// Вмъкване на форматиращи символи в изходния поток данни
ostream& setfld(ostream &str)
{str.width(7);str.precision(3);return str;}

// Печатане на масив
void print(int A[],int N)
{for(int i=0;i<N;i++)
  if((i+1)%10==0)cout<<setfld<<A[i]<<endl;
  else cout<<setfld<<A[i];cout<<endl;
}

void main()
{TAr A,B;int N;
 cout<<"Enter the number of elements:";
 cin>>N;
 GenArr(B,N);
 cout<<"Selection Sort: \n";
```

```
 CpyArr(A,B,N);
 cout<<"Unsorted array:"<<endl;
 print(A,N);
 cout<<"Sorted array:"<<endl;
 SelSort(A,N);
 print(A,N);
 k=N;cout<<"Enter a number to search:";
 cin>>q;
 b=DSearch(A,q,k);
 if(b)cout<<"Found at position "<<k<<endl;
 else cout<<"Not found. Should be at position "<<k<<endl;
 cin>>q;
#ifdef TEST
 cout<<"Checks: "<<CNTP<<" Swaps: "<<CNTA<<endl<<endl;
#endif
 cout<<"Insertion Sort: \n";
 CpyArr(A,B,N);
 cout<<"Unsorted array:"<<endl;
 print(A,N);
 cout<<"Sorted array:"<<endl;
 InsSort(A,N);
 print(A,N);
#ifdef TEST
 cout<<"Checks: "<<CNTP<<" Swaps: "<<CNTA<<endl<<endl;
#endif
 cout<<"Bubble Sort:\n ";
 CpyArr(A,B,N);
 cout<<"Unsorted array:"<<endl;
 print(A,N);
 cout<<"Sorted array:"<<endl;
 BblSort(A,N);
 print(A,N);
#ifdef TEST
 cout<<"Checks: "<<CNTP<<" Swaps: "<<CNTA<<endl<<endl;
#endif
 cout<<"Shake Sort:\n ";
 CpyArr(A,B,N);
 cout<<"Unsorted array:"<<endl;
// print(A,N);
 cout<<"Sorted array:"<<endl;
 ShkSort(A,N);
 print(A,N);
#ifdef TEST
 cout<<"Checks: "<<CNTP<<" Swaps: "<<CNTA<<endl<<endl;
#endif
 cout<<"Improved bubble Sort:\n ";
 CpyArr(A,B,N);
 cout<<"Unsorted array:"<<endl;
 print(A,N);
 cout<<"Sorted array:"<<endl;
 BImSort(A,N);
 print(A,N);
#ifdef TEST
 cout<<"Checks: "<<CNTP<<" Swaps: "<<CNTA<<endl;
#endif
}
```

2.12 Претърсване

Прилага се функция за дихотомично търсене в нареден едномерен масив. При успех функцията връща TRUE, като в третия параметър се връща индексът

на намерената стойност в масива. При неуспех се връщат FALSE и индексът, където би трябвало да се намира търсената стойност. С това функцията би могла да послужи за подобрение на алгоритъма за сортиране чрез пряко вмъкване.

```
program search;
{$APPTYPE CONSOLE}
uses  SysUtils;

const sz=10000;
type T=integer; // Тук може да се впише произволен тип, за който има
                // дефинирани операциите = и <.
     TAr=array[0..sz-1]of T; //Масивът е от същия тип
// Дихотомично търсене. Търси се стойността X в масива A.
// Параметърът N задава размера на масива, а при изход от функцията
// съдържа индекса на намерения елемент или на място за вмъкване.
function DSearch(var A:TAr; X:T; var N:integer):boolean;
var L,R,M:integer;found:boolean;
begin
 L:=0;R:=N-1;found:=false;
 repeat
  M:=(L+R) div 2;
  if A[M]=X then found:=true
  else
  if A[M]<X then L:=M+1 else R:=M-1;
 until found or (L>R);
 if found then N:=M else N:=L;
 DSearch:=found;
end;

procedure SelSort(var A:TAr; N:integer);
var i,j,imin:integer;
    min:T;
begin
for i:=0 to N-2 do
 begin
  imin:=i;min:=A[imin];
  for j:=i+1 to N-1 do
  begin
   if A[j]<min then
   begin min:=A[j];imin:=j;end;
  end;
  A[imin]:=A[i];A[i]:=min;
end;
end;


procedure GenArr(var A:TAr;N:integer);
var i:integer;
begin for i:=0 to N-1 do A[i]:=random(1000);end;

procedure PrnArr(var A:TAr;N:integer);
var i:integer;
begin
 for i:=0 to N-1 do
  if (i+1) mod 10 =0 then writeln(A[i]:7)else write(A[i]:7);
  writeln;
end;
```

```
var A:Tar;N,k:integer;b:boolean;
begin
 write('Enter the number of elements:');
 readln(N);
 GenArr(B,N);
 SelSort(A,N);
 PrnArr(A,N);
 k:=N;
 write('Enter a number to search:';
 readln(q);
 b:=DSearch(A,q,k);
 if b then writeln('Found at position ',k)
 else writeln('Not found. Should be at position ',k);
 readln;
end.
```

```
#include <iostream.h>
#include <cstdlib>
#define TEST
//#undef TEST
const int sz=10000;

typedef int T;
typedef T TAr[sz];

// Дихотомично търсене. Търси се стойността X в масива A.
// Параметърът N задава размера на масива, а при изход от функцията
// съдържа индекса на намерения елемент или на място за вмъкване.
bool DSearch(TAr A, T X,int &N)
{int L=0,R=N-1,M;
 do
 {M=(L+R)/2;
 if(A[M]==X){N=M;return true;}
 if(A[M]<X)L=M+1;else R=M-1;
 }while(L<=R);
 N=L;return false;
}

void SelSort(TAr A,int N)
{int i,j,imin;
 T min;
for (i=0;i<N-1;i++)
{imin=i;min=A[imin];
  for(j=i+1;j<N;j++)
  if(A[j]<min)
   {min=A[j];imin=j;}
  A[imin]=A[i];A[i]=min;
}
}
void GenArr(int A[],int N)
{for (int i=0;i<N;i++)A[i]=rand()%1000;}

ostream& setfld(ostream &str)
{str.width(7);str.precision(3);return str;}

void print(int A[],int N)
{for(int i=0;i<N;i++)
  if((i+1)%10==0)cout<<setfld<<A[i]<<endl;
  else cout<<setfld<<A[i];cout<<endl;
}

void main()
{TAr A;int N,k;bool b;T q;
 cout<<"Enter the number of elements:";
```

```
 cin>>N;
 GenArr(A,N);
 SelSort(A,N);
 print(A,N);
 k=N;cout<<"Enter a number to search:";
 cin>>q;
 b=DSearch(A,q,k);
 if(b)cout<<"Found at position "<<k<<endl;
 else cout<<"Not found. Should be at position "<<k<<endl;
 cin>>q;
}
```

Препоръчителна и лесно достъпна в страната помощна литература:

Основен източник:

1. Wirth N. *Algorithms + data structures = programs*. Prentice-Hall, Englewood Cliffs, N. J. 1973, както и всички издания в превод на български, например:
2. Уирт Н. *Алгоритми + структури данни = програми*. Техника, 1980.

Спомагателни източници:

1. Aho A., Hopscroft J., Ullman J., *Data Structures and Algotithms* Addison-Wesley Publishing Company, 1983
2. Ben-Ari M. *Understanding Programming Languages*, WILEY. 1998, както и в превод на руски:
3. Бен-Ари М. *Языки Програмирования - Практический Анализ.* Москва, Мир, 2000.
4. Манев К. *Увод в дискретната математика.* Издателство на Нов Български Университет, София, 1996.
5. Славов В. *Увод в алгоритмите.* Издателство на Нов Български Университет, София, 1995.
6. Knuth D., *The art of computer programming, Volumes 1,2 and 3* Addison-Wesley Publishing Company, 1973, както и всички издания в превод на руски, например:
7. Кнут Д. *Исскуство программирования для ЭВМ.* Том 1, 2 и 3. Москва, Мир, 1978.
8. Майер Б. Бодуэн К. *Методы програмирования, Том 1 и 2* ., превод от френски, Москва, Мир, 1982.
9. Хорстман Кай, *Принципи на програмирането със C++,* ИК СофТех, С., 2000.

Всички учебници за ВУЗ и други учебни помагала по програмиране на Pascal или C.