Упражнението включва:

1. Бройни системи:

Бройните системи, представляват позиционни и не позиционни бройни системи.

Пример за не позиционна бройна система е римската. В нея всеки знак има, носи някакво значение, но поставянето му на различни места в числото, не променя значението му.

Пример: Ако имаме III – римското число 3, то се изгражда от три римски единици.

Позиционните бройни системи се отличават с това, че всеки символ носи значение спрямо това с какъв знак се отбелязва, но също така получава значение и от това на кое място в числото е поставен. Различните позиции на елементите в записа на числото представляват различни теглови коефициенти.

Пример:

1) Както в горния пример, така и тук ще напишем число от три единици. Но числото ще бъде в десетична бройна система (арабска). Полученото число е 111, но за разлика от горния случай в, който това се равнява на „три" то тук символите на числото получават значение и спрямо това къде са поставени. За първия символ се получава „сто", за втория „десет" и за третия „едно", при което цялото число е „сто и единадесет".

2) Използвайки числото 123, ще разгледаме какво представлява тегловият коефициент. В това число са използвани три цифри: цифрата „3" има теглови коефициент 1-ца, защото е на последно място, цифрата „2" има теглови коефициент 2-ка, защото е на пред последно място, цифрата „1" има теглови коефициент „3-ка", защото е на пред-пред-последно място (тоест тегловият коефициент отговаря на позицията на цифрата в числото, ако броим от ляво на дясно). По аналогичен начин може да продължите с по-големи числа.

Не позиционните бройни системи не представляват голям интерес в изчислителните машини и компютърните системи, поради което разглеждането им по този предмет спира дотук. Основно се използват позиционни бройни системи и поради това ще обърнем повече внимание на работата със тях.

Позиционните бройните системи имат определена основа. Бройната система може да има всякаква основа, като най-често използвани са: 2,8,10,16. Но това не ни пречи да използваме 3,5,7,11 и други екзотични възможности. Преминаването от една в друга бройна система зависи от това, какви са основите на двете бройни системи. Има няколко основни правила:

- Когато се преминава от десетична в друг вид бройна система, цялата част на числото се дели на основата на втората (другата) бройна система, а дробната част на числото се умножава с основата на втората (другата) бройна система.
- Когато се преминава от друг вид бройна система към десетична елементите на числото се умножават по размера на основата на (първата) бройната система (от която се преминава) повдигната на степен. Като степента се определя от тегловия коефициент минус единица.

Примери:

- преминаване от десетична в двойна бройна система.

13(10) -> X(2), 8(10) -> A(2), 63(10) -> C(2), 0,625(10) -> T(2),
27(10) -> Y(2), 19(10) -> B(2), 65(10) -> E(2), 11,45(10) -> R(2).

13 : 2 = 6 + ост(1)
6 : 2 = 3 + ост(0)
3 : 2 = 1 + ост(1)
1 : 2 = 0 + ост(1)

Получените остатаци с взимат в ред обратен на получаването им и се формира следното двойно число: 1101. Следователно 13(10) -> 1101(2), X =1101.

27 :2 = 13 + ост(1)
13 :2 = 6 + ост(1)
... (повтаряйки горния пример)

От делението на 27 при което се получава „13" и остатък „1-ца" може да използваме получения резултата който вече имаме за делението на 13 и към стария резултат ще добавим още една единица най-отпред. Следователно 27(10) -> 11101(2), Y(2) =11101.

8 : 2 = 4 + ост(0)
4 : 2 = 2 + ост(0)
2 : 2 = 1 + ост(0)
1 : 2 = 0 + ост(1) Следователно 8(10) -> 1000(2), A(2) =1000.

19 : 2 = 9 + ост(1)
9 : 2 = 4 + ост(1)
4 : 2 = 2 + ост(0)
2 : 2 = 1 + ост(0)
1 : 2 = 0 + ост(1) Следователно 19(10) -> 10011(2), B(2) =10011.

63 : 2 = 31 + ост(1)
31 : 2 = 15 + ост(1)
15 : 2 = 7 + ост(1)
7 : 2 = 3 + ост(1)
3 : 2 = 1 + ост(1)
1 : 2 = 0 + ост(1) Следователно 63(10) -> 111111(2), C(2) =111111.

65 : 2 = 32 + ост(1)
32 : 2 = 16 + ост(0)
16 : 2 = 8 + ост(0)
8 : 2 = 4 + ост(0)
4 : 2 = 2 + ост(0)
2 : 2 = 1 + ост(0)
1 : 2 = 0 + ост(1) Следователно 63(10) -> 1000001(2), E(2) =1000001.

0,625 . 2 = 1,25 - цяла част = 1, дробна част = 0,25 ,
0,25 . 2 = 0,5 - цяла част = 0, дробна част = 0,5 ,
0,5 . 2 = 1 - цяла част = 1, дробна част = 0.

Когато дробната част стане = 0 тогава приключваме умножението. T(2) = 0,101 (2).

1,45 – цяла част = 1, дробна част = 0,45 ,
0,45 . 2 = 0,9 - цяла част = 0, дробна част = 0,9 ,
0,9 . 2 = 1,8 - цяла част = 1, дробна част = 0,8 ,
0,8 . 2 = 1,6 - цяла част = 1, дробна част = 0,6 ,
0,6 . 2 = 1,2 - цяла част = 1, дробна част = 0,2 ,
0,2 . 2 = 0,4 - цяла част = 0, дробна част = 0,4 ,
0,4 . 2 = 0,8 - цяла част = 0, дробна част = 0,8.
В този пример се появява период. При умножението на дробната част като на ред 2-ри и на последния ред се получава дробна част 0,8 , при което ще се получи зацикляне на умножението. Поради което можем да го запишем до този знак. Получената дробна част е 0,011100 цялото число е 1, и резултатът е 0,011100 или 0,0111. R(2) = 0,0111(2).

Примери за преминаване от двоична в десетична бройна система.
$101_{(2)}$ -> $G_{(10)}$, $1010101_{(2)}$ -> $Y_{(10)}$, $10111111_{(2)}$ -> $Q_{(10)}$,
$10001_{(2)}$ -> $U_{(10)}$, $1001001_{(2)}$ -> $J_{(10)}$, $11111101_{(2)}$ -> $P_{(10)}$.

$101_{(2)} => 1 . 2^2 + 0 . 2^1 + 1 . 2^0 = 1 . 4 + 0 . 2 + 1 . 1 = 4 + 1 = 5,$ $G_{(10)} = 5$
$10001_{(2)} => 1.2^4 + 0.2^3 + 0.2^2 + 0.2^1 + 1.2^0 = 1.16+1.1 = 16 + 1 = 17$ $U_{(10)} = 17$
$1010101_{(2)} => 1.2^6 + 0.2^5 + 1.2^4 + 0.2^3 + 1.2^2 + 0.2^1 + 1.2^0 = 1.64 + 1.16 + 4 + 1 = 85$ $Y_{(10)} = 85$
$1001001_{(2)} => 1.2^6 + 0.2^5 + 0.2^4 + 1.2^3 + 0.2^2 + 0.2^1 + 1.2^0 = 1.64 + 1.8 + 1.1 = 73$ $J_{(10)} = 73$
$10111111_{(2)} => 1. 2^7 + 0.2^6 + 1.2^5 + 1.2^4 + 1.2^3 + 1.2^2 + 1.2^1 + 1.2^0 = 128+32+16+8+4+2+1 = 191 = Q_{(10)}$
$11111101_{(2)} => 1. 2^7 + 0.2^6 + 1.2^5 + 1.2^4 + 1.2^3 + 1.2^2 + 1.2^1 + 1.2^0 = 128+64+32+16+8+4+1= 253 = P_{(10)}$

Когато целите десетични числа са отрицателни и ние искаме да ги превърнем в двоични отрицателни числа се използват следните три стъпки:
1- Взима се реалната част на числото, тя се представя като положителна и се превръща към двоична бройна система.
2- Всички цифри „0” и „1” се заменят съответно с „1” и „0”. Тоест 1 -> 0, 0 -> 1. Правейки това се получава обратния код на положителното число.
3- Полученото число сумираме с една единица „1”. По този начин обратния код се превръща в допълнителен код. Най-отпред на числото се добавя един знаков бид, който за положителни числа е „0” а за отрицателни е „1”. Тоест в нашия случай добавяме една единица в началото на числото (от ляво) и получаваме крайния вариант на отрицателното цяло двоично число.
Примери:
-13(10) = L(2), -17(10) = I(2), -31(10) = W(2).

1) -13 -> 13(10) = 1101(2)
2) Стойностите се инвертират - > 1101 -> 0010 – получава се обратен код
3) 0010 + 1 = 0011 = допълнителен код . L = 1 0011 = крайно число с знаков разряд „1-ца”
1) -17 -> 17(10) = 10001(2)
2) Стойностите се инвертират - > 10001 -> 01110 – получава се обратен код
3) 01110 + 1 = 01111 = допълнителен код .

1 01111 = крайно число с знаков разряд „1-ца”. I = 1 01111 (2)

1) -31 -> 17(10) = 11111(2)
2) Стойностите се инвертират - > 11111 -> 00000 – получава се обратен код
3) 00000 + 1 = 00001 = допълнителен код .
1 01111 = крайно число с знаков разряд „1-ца”. W = 1 00001 (2)

При записването на Получените стойности е хубаво да се спазва определени правила.
Примерно да записваме числата на по 8 бита тоест на 1 байт.
Примерно числото 13 (10) което се представя като 10001 (2) е добре да се представи
като 0 001 0001. По аналогичен начин:
8 (10) = 1000 (2) = 0 000 1000 -8 = 1 111 1000
7 (10) = 111 (2) = 0 000 0111 -7 = 1 111 1001
11 (10) = 1011 (2) = 0 000 1011 -11 = 1 111 0101
18 (10) = 10010 (2) = 0 001 0010 -18 = 1 110 1110

По този начин се представят числата за който 7 бита са достатъчни за да ги представят и осмият (най-левия) се използва за представяне на знака на числото. Такива числа са в диапазон от -127 до +127. За числа извън този диапазон се използват 16, 32, 64, 128 разряда. Разрядите с който работят компютърните системи се определя от техния процесор, съответно 32, 64, и т.н. битови процесори. При малки числа, които не използват всичките битове празните битови от ляво се запълват с нули за положителните числа и с единици за отрицателните числа.
Примери:

6(10) = 0 110 (2) = 0 000 0110 (2) – байт
0 000 0000 0000 0000 0000 0000 0000 0110 – 4 байта – 32 бита представяне.

-6(10) = 1 010 (2) = 1 111 1010 (2) байт
1 111 1111 1111 1111 1111 1111 1111 1010 – 4 байта – 32 бита представяне.

За представянето за дробни отрицателни числа се използват формати REAL(4) и REAL(16), които ще бъдат разгледани по предмета Организация на Компютрите (ОК) и не са цел на това занятие.

2. Сложност на алгоритмите.
Сложността на алгоритмите се бележи с O().
Имаме 2 вида сложност – по време и по памет. Като сложността по памет се разглежда по-рядко. Също така имаме Асимптотична сложност. Тя е във функция от максималния размер на задачата. Обикновено това се дава като сложност на алгоритъма.

Пример: Ако в един алгоритъм имаме $10N^2 + 18N + 15$ на брой действия то сложността на този алгоритъм се приема за $O(N^2)$ – квадратична сложност.

Имаме различни видове сложности: O(C), O(logN),O(NlogN),$O(N^2)$,$O(N^3)$ и тн.. Тези сложности могат да бъдат добри или лоши зависимост от алгоритмите при които ги получаваме.

$O(N\log_2 N)$ – Бързи алгоритми за сортиране;
$O(N^{1,4\text{-}1,6})$ – Междинна сложност за сортиране;
O(N) – Линейна сложност;
$O(\log_2 N)$ – Бързи алгоритми за търсене;

В случай, че имаме вложени цикли като всеки от циклите зависи от N линейно или чрез определена нелинейна независимост, тогава сложността се умножава (при вложени 2 цикъла for от 1 до N, сложността е N^2). А ако са последователни сложността им се събира.

Примери:

```
For (i=0; i<a; i++)
{   for (i=0; i=b; i++)
   { <тяло на вътрешния цикъл> }
}
```

В този пример сложността е равна на a*b* (<тялото на вътрешния цикъл>) Ако a=b=n тогава сложността ще бъде n^2*(<тялото на вътрешния цикъл>). Ако <тялото на вътрешния цикъл> има сложност O(C) – константна то получаваме n^2*C което може да бъде прието за n^2.

```
for (i=0; i<a; i++)
{ <тяло на първи цикъл> }
for (i=0; i=b; i++)
{ <тяло на втори цикъл> }
```

В този случай циклите са последователни и сложността на фрагмента от код е
A*(<тяло на първи цикъл>)+b*(<тяло на втори цикъл>) . Ако a=b=n и тялото на циклите е с константна сложност тогава O(C). Тогава крайната сложност ще бъде n^2 *C или n^2.

При по сложни алгоритми трябва да се стремим към разделянето на кода на разбираеми и лесни за анализирани фрагменти и след уточняване на сложността на отделните фрагменти да изчислим сложността на целия алгоритъм.

Работа с Масиви.
Масива е най-популярната структура от данни. Основното разбиране е че той е явно достъпна структура, елементите на която са от един и същи тип. В програмите освен масиви се използват и различни записи. Който ще разгледаме в друго упражнение. Също така в програмирането се използват Бази от данни, които предимно се използват в приложения, които съхраняват по-голям брой записи в себе си, но напоследък навлизат все повече дори в малките и леки приложения. Базите от данни също не са предмет на нашето упражнение, като те се разглеждат в отделен предмет.
След този увод нека да се върнем на нашата задача „работа с масиви". Масивите имат основното свойство да имат измерения. Най често ползвани са масиви с едно и две измерения и въпреки че нормалните ни представи са за свят с три измерения, масивите могат да имат теоретично безкраен брой измерения. Какви рискове крият големият брой от измерения ще разгледаме по-надолу. Предназначението на масивите а именно „съхранението на еднотипни данни в големи количества" създава и предпоставки за действията който най-често се извършват върху масивите. Тези действия са: търсене на елемент в масив, обхождане на масив с различни цели, подреждане на стойностите в масив по определен критерии. Има и други свойства но тези са основни. На сортирането на масиви ще отделим повече време когато говорим за различните видове сортиране, а сега ще обърнем внимание на обхождането и търсенето на определен елемент.

Първо да покажем как се дефинира (съставя) един масив. Един масив съдържа тип на променливите, име на масива брой на измеренията и брой на елементите във всяко измерение.

<тип на променливите> <име на масива> [<брой елементи>]; - в случая едномерен.
<тип на променливите> <име на масива> [<брой елементи>] [<брой елементи>]; - двумерен.
Тима на променливите на масива могат да бъдат както стандартни типове така и дефинирани типове от потребителя или дори определени класови структури. Тоест масива може да е масив от елементи на един клас. Името на масива както и имената на променливите трябва да започва с буква и да се състои от определени символи (букви и цифри и ограничен брой допълнителни символи). Броя на измеренията зависи от броя на двойките квадратни скоби, който сме използвали. Като масив с една двойка квадратни скоби представлява едномерен масив, масив с две двойки квадратни скоби е двумерен и т.н.. Броят на елементите в всяко измерение може да бъде произволен. Езикът на който ще реализираме примерите е
C-подобен.

Примери:

int students [26]; - дефинираме едномерен масив с име students съдържащ 26 елемента от тип int.

int students [26] [5] [2]; - дефинираме тримерен масив с име students съдържащ 26 * 5 * 2 елемента от тип int. Като елементите в този масив могат да бъдат възприети като два потока по 5 групи по 26 студенти.

След като разгледахме, как може да се дефинира един масив сега да разгледаме как може да се намери определен елемент в него и как да обходим масив. Примери:
За всички примери ще приемем че масивите вече са изградени и имат произволни стойности.
1) Да обходим едномерен масив и да принтираме неговите елементи.
2) Да обходим двумерен масив и да и да увеличим всяка негова стойност 3 пъти.
3) Да намерим първият срещнат елемент със стойност 6 и да принтираме поредния му номер от последователността на масива.
4) Да преброим колко от елементите на тримерен масив имат стойност 5.

Пр. 1
```
for (i=0; i<N; I++)
{   Printf(" %d  ", MASIV[i]);   }          //  Сложност на алгоритъма N
```

Пр. 2
```
for (i=0; i<N; I++)
{  for (r=0; r<N; r++)
   {
      MASIV[i][r] = (MASIV[i][r]*3);
   }
}                                          // Сложност на алгоритъма N^2
```

Пр. 3
```
while ((i=0) && (i<N))
{
   if (MASIV[i]=6)
   {
      printf("pyrviq sreshtnat element sys stojnost 6 e  %d", i);
   break;
   }
}                                                                  // сложност N
printf("v masiva nqma nito edin element sys stoinost 6");
```

Пр. 4
```
Int temp = 0;        // стойност в която натрупваме броя на срещнатите елементи отговарящи на if
for (i=0; i<N; I++)                  // цикъл за първо измерение на масива
{  for (r=0; r<N; r++)               // цикъл за второ измерение на масива
   {   for (v=0; v<N; v++)           // цикъл за трето измерение на масива
       {  if (MASIV[i][r][v]=5)      // услови на примера
          {  temp = temp + 1;  }     // натрупване при изпълнение на условието
       }
   }
}                                                                  // сложност N^3
printf("broqt na elementite sys stoinost 5 e  %d ", temp);         // извеждане на крайния резултат
```

Във всички тези 4 примера показахме, как се обхождат масиви. Като в последните два от тях показахме, как могат да се търсят определени елементи. Но последните два примера показват търсене в неподредени масиви. При подредените масиви можем директно да сравняваме търсената стойност с един елемент избран от част или от целия масив. Примерно избираме средата на масива и сравняваме с нея, ако тя не съответства с търсената взимаме за отправна точка горната или долната половина от масива и изпълняваме същото... Така повтаряме идеята докато намерим търсения елемент или сведем диапазона за търсене до 1 стойност и ако тя не съвпада значи в подредения масив няма стойност равна на търсената. Масивите имат още един белег. Този белег е наличието на диагонали, но за да имаме диагонали в един масив то броят на елементите му във всяко измерение трябва да е равен. Ако един масив има 1 измерение, в което има 10^5 елемента тоест 100 000 елемента или има 5 имзерения по 10 елемента тоест 100 000 елемента времето за работа с него е еднакво, но ако има (примерно) 5 измерения и оставим потребителя да реши, колко елемента да има измерението то ще имаме зависимост N^5. И ако потребителя зададе ненужно голяма стойност за N, проблемът с N^5 сложността ще е лавинообразен.

Методи за сортиране. Метод на мехурчето, Метод на пряката селекция, Метод чрез вмъкване. Тези три вида сортировки имат сходства и разлики. Примерно и 3-та са със сложност $O(N^2)$. Различията им се явяват от начинът им на работа които ще разгледаме по-надолу. Но основно за тези сортировки е, че те са бавни методи за сортиране.

Метода на мехурчето се характеризира с това, че елементите му се сравняват по два съседни. Този вариант в който двойката сравняващи се елементи се е приела за мехурче което се издига нагоре.

```
void bubbleSort(int numbers[], int array_size)
{int i, j, temp;
 for (i = (array_size - 1); i >= 0; i--)
 {for (j = 1; j <= i; j++)
  {if (numbers[j-1] > numbers[j])
   {temp = numbers[j-1];
    numbers[j-1] = numbers[j];
    numbers[j] = temp;
   }}}}
```

Метода на пряката селекция се характеризира с това чесе избира един елемент и всички останали се сравняват с него докато не се намери елемент отговарящ на поставеното условие (по-голям или по-малък) след което двата елемента се разменят и сравнението продължава. Разликата между този метод и метода на мехурчето е че тук един избран елемент се сравнява с всички останали (под или над него) а не само с още един елемент.

```
void selectionSort(int numbers[], int array_size)
{int i, j;
 int min, temp;

 for (i = 0; i < array_size-1; i++)
 {min = i;
  for (j = i+1; j < array_size; j++)
  {if (numbers[j] < numbers[min])
     min = j;
  }
  temp = numbers[i];
  numbers[i] = numbers[min];
  numbers[min] = temp;
 }}
```

Метода на вмъкването е сходен с метода на пряката селекция. Отново един избран елемент се сравнява с всички останали но когато се намери елемент отговарящ на условието на размяна не се разменят на момента а се проверява и следващия и се ако е възможно цели пасажи се преместват а началния избран елемент се поставя (под или над) тях спрямо условието.

```
void insertionSort(int numbers[], int array_size)
{int i, j, temp;
 for (i=1; i < array_size; i++)
 {temp = numbers[i];
  j = i;
  while ((j > 0) && (numbers[j-1] > temp))
   {numbers[j] = numbers[j-1];
    j = j - 1;
   }
  numbers[j] = temp;
}}
```

Препоръчва се да се запомни начинът на работа на алгоритмите вместо да се наизустява кода без разбиране.
Сайтове в които можете да намерите материали са:
1. http://www.google.bg/ 2. http://bg.wikipedia.org/
3. http://www.sorting-algorithms.com/ - Хубав сайт с анимации на действието на методите за сортиране.
http://193.192.57.240/po/courses/2009/elektronenKursPoProgramirane-1/pdf/ - различни примри.
КАКТО И ДРУГИ ... не се ограничавайте в търсенето на знания!