{

Учебно помагало по

АЛГОРИТМИ И ПРОГРАМИ

със C#

Ангел Голев

}

Пловдив, 2012

Пловдивски университет „Паисий Хилендарски”
Факултет по математика и информатика

Ангел Голев

Учебно помагало по алгоритми и програми със C#

Рецензент: проф. д-р Асен Рахнев

Университетско издателство „Паисий Хилендарски”

Пловдив, 2012

ISBN 978-954-423-791-2

Съдържание

Въведение

Това ръководство е предназначено за студентите, изучаващи избираемата дисциплина „Алгоритми и програми в състезанията”. Може също да се използва за подготовка на студенти и ученици за състезание, а също и от учители и преподаватели.

Разглеждат се голям брой известни алгоритми, обяснени с конкретни задачи и с приложени реализации. Тези алгоритми и задачи в голяма степен покриват застъпените теми в студентските и ученическите състезания. Целта е студентите да разберат как се съставят алгоритми и как се реализират на практика.

Курсът е предназначен за студенти, които трябва да са запознати с програмирането на **C++** или **C#**. Ако целта на обучението е подготовка за състезание, то средствата на използвания език за програмиране трябва да се познават идеално. Ръководството също може да послужи на студенти, изучавали **C++** да навлязат по-бързо в програмирането със **C#**.

Навсякъде в програмите, с цел за по-компактен и ясен код, се използват класовете от колекцията **System.Collections.Generic**. Използват се класовете за списъци, стекове, опашки, списъци и сортирани списъци с двойка елементи ключ и стойност. В курса не се разглежда реализацията на тези динамични структури. Същите класове са реализирани и в библиотеката **STL** за **C++**, така че програмите могат много лесно да се напишат и на **C++**.

Ръководството може да се използва и за затвърждаване на знанията по програмиране, обектно-ориентирано програмиране и програмиране на **C#**.

1. Рекурсивни функции

Ще разгледаме няколко известни задачи, които могат да се решат с рекурсивни функции.

Пресмятане на факториел

След като имаме рекурентна формула лесно можем да реализираме съответна рекурсивна функция:

$$f(n) = nf(n-1), \quad f(0) = 1, \quad f(1) = 1.$$

```
public static long RecursiveFactorial(int n)
{
    if (n <= 1)
        return 1;
    else
        return n * RecursiveFactorial(n - 1);
}
```

Обикновено не се налага директно пресмятане на ***n!***, а факториелът се получава с натрупване. Например, изчисляването на биномен коефициент с формулата

$$C(n,k) = \frac{n!}{k!(n-k)!} = \frac{(n-k+1)(n-k+2)...n}{1.2...k}$$

става по следния начин:

```
public static int BinomialCoef(int n, int k)
{
    double coef = 1;
    for (int i = n - k + 1, j = 1; j <= k; i++, j++)
        coef = coef * i / j;
    return (int)coef;
}
```

Освен това директното изчисляване на трите факториела в първата дроб може да доведе до препълването на променливите, който използваме и получаването на грешен резултат.

Независимо от езика и средата за програмиране при всяко рекурсивно обръщение се отделя памет за локалните променливи на функцията. Ако броят на последователните рекурсивни обръщения стане много голям, това може да доведе до препълване на стека.

Числа на Фибоначи

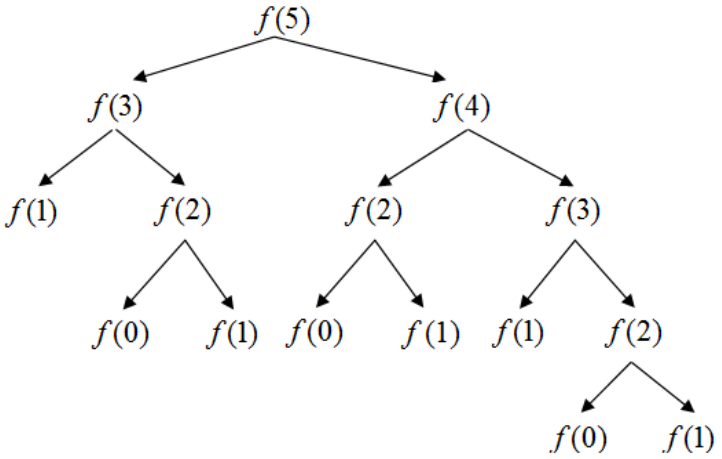
Самото определение на числата на Фибоначи е с рекурентна формула

$$f(n) = f(n-1) + f(n-2), \quad f(0) = 0, \quad f(1) = 1.$$

Реализацията на рекурсивната функция е елементарна

```
public static long FibonacciNums(int n)
{
    if (n <= 1) return n;
    return FibonacciNums(n-1) + FibonacciNums(n-2);
}
```

Но едни и същи числа се изчисляват повторно прекалено много пъти и при голямо ***n*** функцията започва да работи бавно. Обръщенията нарастват експоненциално с нарастването на ***n***.



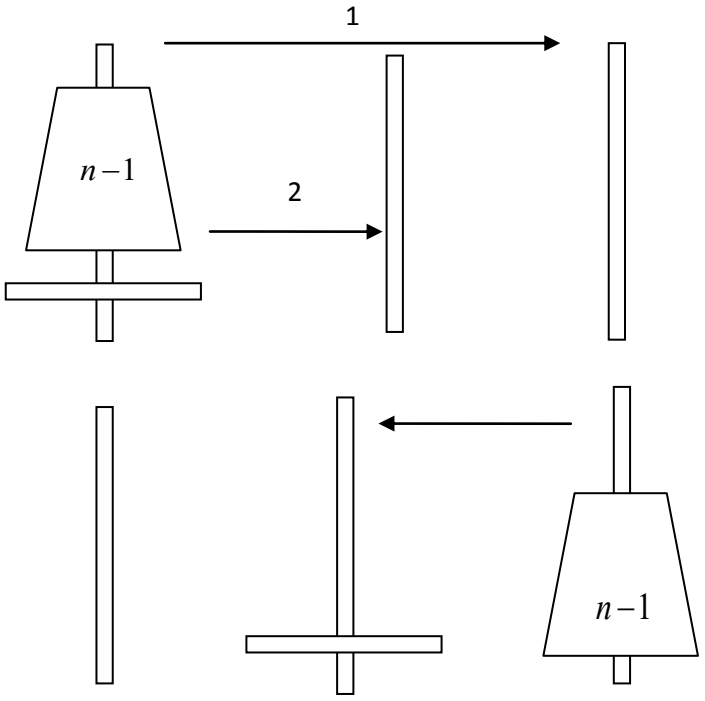
В примера с намирането на петото число на Фибоначи, ***f(3)*** се изчислява два пъти, ***f(2)*** – три пъти, а ***f(1)*** – пет пъти. В случая е най-добре да се използва итеративно изчисляване на функцията, което е с линейна скорост и не използва допълнителна памет.

```
int a = 0, b = 1, c;

for (int i = 2; i <= n; i++)
{
    c = a + b;  a = b;  b = c;
}
return c;
```

Игра ханойска кула

Имаме ***n*** на брой диска, различни по размер един от друг и три стълба, като дисковете могат да се нанизват на стълбовете. В началото дисковете са подредени на единия стълб, като най-големият е най-отдолу, а най-малкият – най-отгоре. Целта е кулата от дискове да бъде преместена на друг стълб. Може да се мести само по един диск на ход и не може по-голям диск да бъде поставен върху по-малък. На всеки ход се взима горния диск от даден стълб и поставя най-отгоре на друг стълб.

Тази задача се решава много лесно и оптимално с рекурсивна функция и значително по-трудно, ако използваме итеративен алгоритъм. Приемаме, че знаем как да преместим ***n-1*** диска от стълб ***1*** на стълб ***2*** с междинен ***3***. Преместваме ***n*** диска от стълб ***1*** на ***2*** по следния начин:



- Преместваме горните ***n-1*** диска от стълб ***1*** на ***3***;
- Преместваме най-големия диск от стълб ***1*** на ***2***;
- Преместваме ***n-1*** диска от стълб ***3*** на ***2***.

Най-големият диск не пречи на преместването на останалите.

```
static void Hanoi(int num_disks, char rod_1, char rod_2, char rod_3)
{
    if (num_disks == 0) return;

    Hanoi(num_disks - 1, rod_1, rod_3, rod_2);
    Console.WriteLine(rod_1 + " -> " + rod_2);
    Hanoi(num_disks - 1, rod_3, rod_2, rod_1);
}

static void Main(string[] args)
{
    Hanoi(4, '1', '2', '3');

    Console.ReadKey();
}
```

По-нататък са разгледани и обяснени и други рекурсивни алгоритми.

Няма правила за избор дали да използваме рекурсивни или итеративни алгоритми. Рекурсивните алгоритми могат да доведат до по-кратки и елегантни решения. Можем да ги използваме, ако реализираме известни алгоритми или алгоритъмът е напълно ясен на програмиста. В противен случай рискуваме програмата да не дава резултата, който очакваме, а рекурсивните функции трудно се проследяват по време на работа.

2. Числени задачи

2.1. Намиране на прости числа

Ще разгледаме няколко подобряващи се варианта за намиране на прости числа. По определение просто число е естествено число по-голямо от ***1***, което се дели само на ***1*** и на себе си. Ще напишем програма, която проверява дали въведено естествено число ***num*** е просто. Най-простата реализация е да проверим дали числото не се дели на числата от ***2*** до ***num-1***. Следва кода на програмата:

```
static void Main(string[] args)
{
    int num,i;
    Console.Write("Enter a number: ");
    num = int.Parse(Console.ReadLine());

    for (i=2; i<num; i++)
        if (num % i == 0)
        {
            Console.WriteLine("The number "+num+" isn't prime.");
            break;
        }
    if (i == num) Console.WriteLine("The number "+num+" is prime");

    Console.ReadKey();
}
```

В предложения алгоритъм можем да намалим броя на проверките за делимост на половина, тъй като едно число не може да се дели на число по-голямо от неговата половина. Още по-силно условие е, че числото се дели най-много на корен квадратен от него. За функцията корен квадратен **Sqrt()** използваме класа **System.Math**.

```
static void Main(string[] args)
{
    int num, i;
    Console.Write("Enter a number: ");
    num = int.Parse(Console.ReadLine());

    for (i = 2; i <= Math.Sqrt(num); i++)
        if (num % i == 0)
        {
            Console.WriteLine("The number isn't prime");
            break;
        }
    if (i > Math.Sqrt(num)) Console.WriteLine("The number is prime");

    Console.ReadKey();
}
```

Това решение може да се използва, когато трябва да намерим простите числа в някакъв предварително зададен интервал, т.е. проверяваме всяко число поотделно. Когато левият край на интервала е близко до ***2*** можем да използваме още два метода. Първият се нарича *решето на Ератостен*: Имаме числата от ***2*** до ***num*** и искаме да намерим всички прости числа в този интервал. Започваме с ***2*** и зачертаваме всички числа, които се делят на ***2*** или всяко второ число от ***2*** нататък. След това намираме първото незачертано число след ***2***, което е следващото просто, в случая е ***3*** и зачертаваме всяко ***3***-то число до края. На всяка следваща стъпка намираме следващото просто число ***p***, което не е зачертано, и зачертаваме всяко ***p***-то число до края. След приключване на процедурата незачертаните числа в интервала са прости. За зачертаването (маркирането) на числата използваме логически масив ***number*** с дължина ***num***:

```
static bool[] number;

static void Main(string[] args)
{
    int num, i, j;
    Console.Write("Enter a number: ");
    num = int.Parse(Console.ReadLine());

    number = new bool[num+1];
    number[1] = false;
    for (i = 2; i <= num; i++) number[i] = true;

    Console.WriteLine("The prime numbers less or equal than "
                      + num.ToString() + " are:");
    for (i = 2; i <= num; i++)
        if (number[i])
        {
            Console.Write(i + ", ");
            for (j = 2*i; j <= num; j += i) number[j] = false;
        }
    Console.ReadKey();
}
```

Другият начин за намиране на всички прости числа по-малки от зададено е следният: за всяко следващо число ***k*** от зададения интервал проверяваме дали ***k*** се дели на простите числа, които са намерени до момента. Ако числото ***k*** е просто, го добавяме в списъка от прости числа и продължаваме нататък. В програмата използваме списък от тип **List<int>**, за да съхраним намерените до момента прости числа:

```
static void Main(string[] args)
{
    int num, i;
    Console.Write("Enter a number: ");
```

```
        num = int.Parse(Console.ReadLine());

        Console.WriteLine("The prime numbers less or equal than "
                          + num.ToString() + " are:");

        List <int> primes = new List<int>() { 2, 3 };
        bool prime_flag;
        for (i = 4; i <= num; i++)
        {
            prime_flag = true;
            foreach (int p in primes)
                if (i % p == 0)
                {
                    prime_flag = false;  break;
                }
            if (prime_flag) primes.Add(i);
        }
        foreach (int p in primes) Console.Write(p + ", ");

        Console.ReadKey();
    }
```

Този вариант е по-бърз от метода на Ератостен и използва по-малко памет.

А с помощта на библиотеката **LINQ** и използването на ламбда израз основните цикли в програмата могат да изглеждат по следния начин:

```
        for (i = 4; i <= num; i++)
        {
            if (! primes.Any(p => i % p == 0)) primes.Add(i);
        }
```

2.2. Най-голям общ делител, най-малко общо кратно, разлагане на естествено число на прости множители

Най-голям общ делител

Намирането на най-голям общ делител (НОД) се налага често в различни числови задачи. В случая ще реализираме метода на Евклид за намиране на НОД. Имаме две числа a и b. Намираме остатъка res от делението на a на b. На следващия ход делимото a става равно на b, а делителят b – на res. Продължаваме да намираме остатъците при делението по същия начин, докато остатъкът стане равен на 0. Най-големият общ делител е равен на последния ненулев остатък.

```
static int GreatCommonDivisor(int a, int b)
{
    int rest;
    do
    {
        rest = a % b;
        a = b;
        b = rest;
    }
    while (rest > 0);
    return a;
}
```

Може да се използва и варианта с изчерпващо изваждане, който се предлага в почти всички книги. Предимството му е в по-лесно запомнящия се код. За по-бърз код е спорно да се говори, тъй като в съвременните процесори събирането и изваждането не са много по-бързи от операциите за умножение, деление и остатък.

```
static int GreatCommonDivisor2(int a, int b)
{
    while (a > 0 && b > 0)
    {
        if (a > b)
            a -= b;
        else
            b -= a;
    }
    return a > 0 ? a : b;
}
```

Най-малко общо кратно

За намирането на най-малкото общо кратно използваме формулата:

$$НОК(a,b) = \frac{a.b}{НОД(a,b)}$$

Разлагане на естествено число на прости множители

Имаме зададено число ***num***. Трябва да разложим числото на прости множители по следния начин:

$$num = k_1^{p_1}.k_2^{p_2}\cdots k_n^{p_n}$$

За всяко число ***k*** от ***2*** до корен квадратен от ***num*** извършваме следната процедура: проверяваме дали ***k*** дели ***num***, ако не се дели, проверяваме със следващото число, ако ***k*** дели ***num***, имаме множител ***k*** и трябва да проверим на каква степен ***k*** се включва в представянето на числото, т.е. премахваме множителя ***k*** от ***num*** и това правим с разделяне на ***num*** на ***k***,

докато ***num*** повече не се дели на ***k***. Текущият множител ***k*** и степента му ***p*** записваме в списък.

```
public struct mult_pow
{
    public int multiplier;
    public int power;
}

static void Factorization()
{
    int num, k, p;
    bool prime_num;
    Console.Write("num="); num = int.Parse(Console.ReadLine());
    List<mult_pow> factor = new List<mult_pow>();

    k = 1;
    while (num > 1)
    {
        k++;
        prime_num = true;
        while (k <= Math.Sqrt(num))
            if (num % k == 0)
            {
                prime_num = false;
                break;
            }
            else
                k++;
        if (prime_num)
        {
            factor.Add(new mult_pow() { multiplier = num, power = 1 });
            break;
        }
        p = 0;
        do
        {
            num /= k;
            p++;
        }
        while (num % k == 0);
        factor.Add(new mult_pow() { multiplier = k, power = p });
    }

    bool fstep = true;
    foreach (mult_pow mpe in factor)
    {
        if (fstep) fstep = false; else Console.Write('*');
        Console.Write(mpe.multiplier);
        if (mpe.power > 1)
            Console.Write("^" + mpe.power);
    }
    Console.WriteLine();
}
```

Ако разполагаме с предварително намерени прости числа, можем да ги използваме, като проверяваме за делимост с тях, а след като проверим и с най-голямото от тях, започваме да проверяваме за делимост с по-големи последователни естествени числа.

След разлагането на две числа на прости множители,

$$a = k_1^{p_1} k_2^{p_2} \cdots k_n^{p_n}; \qquad p_i \geq 0, i = 1, n$$
$$b = k_1^{q_1} k_2^{q_2} \cdots k_n^{q_n}; \qquad q_i \geq 0, i = 1, n$$

можем да намерим НОД и НОК по следния начин:

$$НОД(a.b) = k_1^{\min(p_1,q_1)} k_2^{\min(p_2,q_2)} \cdots k_n^{\min(p_n,q_n)}$$

$$НОК(a.b) = k_1^{\max(p_1,q_1)} k_2^{\max(p_2,q_2)} \cdots k_n^{\max(p_n,q_n)}$$

2.3. Представяне на числа в различни бройни системи

Преобразуване на число от десетична в p-ична бройна система

За да преобразуваме едно число от десетична бройна система в бройна система с основа ***p*** използваме следния алгоритъм: Започваме да делим числото на основата ***p*** с частно и остатък. Остатъците добавяме в списък. След това частното става изходното число и продължаваме да делим на ***p***, докато частното стане равно на ***0***. Обръщаме списъка с остатъците обратно и това са цифрите на числото в ***p***-ична бройна система. Когато основата на новата бройна система е по-голяма от ***10***, за цифри използваме последователно големите латински букви, като ***A=10***, ***B=11, C=12*** и т.н. Буквите получаваме, като към буквата ***A*** добавяме остатъкът минус ***10*** и получаваме (кода на) съответната буква. За целите числа в десетична бройна система използваме типа **long**, а резултатът се записва в списък от символи.

```
static void Num10toOtherBasis()
{
    long num10;
    Console.Write("Number: ");
    num10 = Int64.Parse(Console.ReadLine());

    int basis;
    Console.Write("Basis: ");
    basis = int.Parse(Console.ReadLine());

    List<char> num_b = new List<char>();
    long priv, rem;
```

```
    do
    {
        priv = num10 / basis;
        rem = num10 % basis;
        num_b.Add((char)(rem < 10 ? '0' + rem : 'A' + rem - 10));
        num10 = priv;
    }
    while (num10 > 0);

    Console.Write("The number in new basis is: ");
    num_b.Reverse();
    foreach (char ch in num_b) Console.Write(ch);
    Console.WriteLine();
}
```

Преобразуване на число от q-ична в десетична бройна система

Използваме, че числото $a=(a_k a_{k-1}\ldots a_1 a_0)_p$, записано в ***p***-ична бройна система е равно на $a_k p^k + a_{k-1}p^{k-1} + \cdots + a_1 p + a_0$, където $a_i = 0, n-1$ са цифрите на числото. Директното пресмятане на горния полином е бавно и затова използваме *схемата на Хорнер*:

$$a = \Big(\cdots\Big(\big(a_k p + a_{k-1}\big)p + a_{k-2}\Big)p + \cdots + a_1\Big)p + a_0,$$

в която се използват ***k*** на брой умножения и ***k*** на брой събирания.

Цифрите на числото в ***p***-ична бройна система са записани в низ, а резултатът получаваме като **long**.

```
static long Num10fromOtherBasis(string num_b, int basis)
{
    long num10 = 0;
    foreach (char ch in num_b)
        num10 = num10 * basis + (int)(ch>='A' ? ch-'A'+10 : ch - '0');
    return num10;
}
```

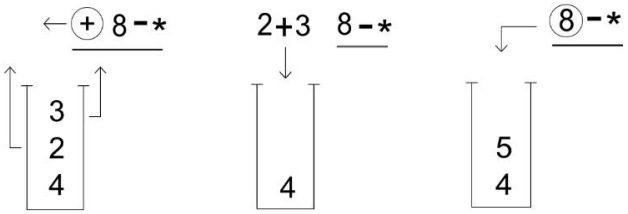
Обратен полски запис

Постфиксен запис (*Postfix Notation*) или обратен полски запис (*Reverse Polish Notation*) е представяне на израз, в което операцията се записва след двата операнда. Удобството на записът е, че не се налага да се използват скоби, за да се укаже приоритета и реда на изпълнение на операциите. Например изразът ***(2+3)*8*** ще се представи в постфиксен запис, като ***2 3 + 8 ****. В този запис се налага да се използва друг знак за унарния минус, например '**~**'. Реализирането на изчисляване на израз записан в постфиксен запис е доста по-лесно от това на инфиксен запис.

Няколко примера за преобразуване на израз в постфиксен запис:

```
(3 + 5) * 9          ⟶      3 5 + 9 *
3 + 5 * 9            ⟶      3 5 9 * +
3 * (5 + 9)          ⟶      3 5 9 + *
3 * 5 + 9            ⟶      3 5 * 9 +
2 - 3 + 4 - 5        ⟶      2 3 - 4 + 5 -
-8 * (a - b)         ⟶      8 ~ a b - *
```

За изчисляване на израз в постфиксен запис използваме следния алгоритъм: Последователно от израза се взимат операнд (числова стойност или стойност на променлива) или операция. Операндите се добавят в стек по реда на пристигането им. Когато от израза се вземе двуаргументна операция, от стека се взимат две стойности в обратен ред, операцията се изпълнява върху тях и резултатът се записва отново в стека. При едноаргументна операция (напр. унарен минус) от стека се взима само една стойност, операцията се изпълнява с тази стойност и резултата отново се записва в стека. Когато се изпълни и последната операция, в стека трябва да има само един елемент и това е пресметнатата стойност на израза.

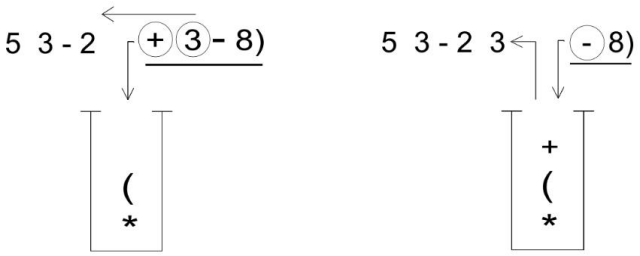


За опростяване на реализацията използваме вместо числа и променливи само едноцифрени числа и операции **+, -, *, /, ~**. Изразът в постфиксен запис е записан в низ, а резултатът е цяло число. Използваме стека от колекцията **System.Collections.Generic** и методите **Pop()** и **Push()**, за взимане на елемент от върха на стека и добавяне на нов елемент към стека.

```
static int CalcPolishNotation(string pn_equation)
{
    int a, b;
    Stack<int> working_stack = new Stack<int>();

    foreach (char ch in pn_equation)
    {
        switch (ch)
        {
            case '*':
            case '/':
            case '+':
            case '-':
```

```
                b = working_stack.Pop();
                a = working_stack.Pop();
                switch (ch)
                {
                    case '*': a *= b; break;
                    case '/': a /= b; break;
                    case '-': a -= b; break;
                    case '+': a += b; break;
                }
                working_stack.Push(a);
                break;
            case '~':                           // унарен минус
                a = working_stack.Pop();
                working_stack.Push(-a);
                break;
            default:
                if (char.IsDigit(ch))
                    working_stack.Push(ch - '0');
                break;
        }
    }
    return working_stack.Pop();
}
```

Преобразуването на израз от инфиксен запис в постфиксен запис става по следния начин: На всеки ход от израза в инфиксен запис взимаме по един елемент. Ако елементът е операнд (число или променлива), добавяме този елемент към списък (това ще е разултатът в постфиксния запис). Ако елементът е скоба или операция използваме помощен стек за временното им съхраняване.



Има разлика при обработването на различните операциите и скобите. Ако елементът е:

- отваряща скоба – вмъкваме скобата в стека;
- затваряща скоба – всички операции, записани в стека до първа отваряща скоба, се изваждат от стека и в реда на изваждане се добавят към резултатния списък, а отварящата скоба се маха от стека без да се записва в резултата;

- операции ***** и **/** – тези операции „избутват” всички операции *****, **/**, **~** до края на стека или до отваряща скоба. Изваденитe (избутани) операции от стека, се записват в същия ред в списъка „резултат”;
- операции **+** и **–** избутват всички останали операции, до отваряща скоба или докато стекът не е празен.

Накрая стекът трябва да е празен, а списъкът съдържа израза в посфиксен (обратен полски) запис.

В реализацията за операнди се използват само цифри. Използва се и методът **Peek()**, който взима елемента от върха на стека, но без да го изважда от стека. Използва се и логически флаг, за да се определи кога един минус е унарен:

```
static string InfixToPostfixNotation(string inf_equation)
{
    string result = "";
    char w_ch;
    Stack<char> working_stack = new Stack<char>();
    bool unary_minus = true;

    foreach (char ch in inf_equation)
    {
        switch (ch)
        {
            case '*':
            case '/':
            // операциите * и / избутват всички * , / и ~ от стека
                while(working_stack.Count > 0 &&
                      ( (w_ch = working_stack.Peek()) == '*' ||
                         w_ch == '/' || w_ch == '~' ) )
                {
                    result += working_stack.Pop().ToString() + ' ';
                }
                working_stack.Push(ch); unary_minus = true;
                break;
            case '+':
            case '-':
            // операциите + и - избутват всички операции до '('
                if (unary_minus)
                {
                    working_stack.Push('~');
                    break;
                }
                while(working_stack.Count > 0 &&
                      (w_ch = working_stack.Peek()) != '(' )
                {
                    result += working_stack.Pop().ToString() + ' ';
                }
                working_stack.Push(ch); unary_minus = true;
                break;
```

```
                case '(':
                    working_stack.Push(ch); unary_minus = true;
                    break;
                case ')':
                // избутва всички операции от стека до '('
                    while ((w_ch = working_stack.Pop()) != '(')
                        result += w_ch.ToString() + ' ';
                    unary_minus = true;
                    break;
                default:
                    if (char.IsDigit(ch))
                    {
                        result += ch.ToString() + ' ';
                        unary_minus = false;
                    }
                    break;
            }
        }
        while (working_stack.Count > 0)
            result += working_stack.Pop().ToString() + ' ';

        return result;
    }
```

2.4. Реализация на събиране и умножение на дълги цели числа

В доста задачи от практиката се налага да се използват цели числа, които са доста големи от тези, за които се използват стандартни типове. В последните версии на **.NET** се предлага класа **BigInteger,** който е изключително бърз и лесен за използване и позволява записването на цели числа с практически произволна дължина. Независимо от това предлагаме една реализация на събиране и умножение с дълги цели числа. Всяка десетична цифра на дългото число се запазва в отделен елемент на масив или списък. В нашия случай използваме списък от тип **byte.**

```
    class LongNum
    {
        List<byte> lnum;

        public LongNum()
        {
            this.lnum = new List<byte>();
        }
        private LongNum(string s, int n)
        {
            this.lnum = new List<byte>(n);
        }
        public LongNum(string s)
        {
            this.Set(s);
        }
```

```
public int Length
{
    get { return this.lnum.Count; }
}

public void Set(string s)
{
    this.lnum = new List<byte>(s.Length);
    for (int i = s.Length-1; i>=0; i--) lnum.Add((byte)(s[i]-'0'));
}

public void Set(long m)
{
    this.Set(m.ToString());
}
```

Събирането на две дълги числа се прави по същия начин, както се събират цели числа на „ръка”. Първо се събират най-десните цифри на числата, ако има пренос ***c***, то той се добавя към следващото число и продължаваме по същия начин да събираме всяка следваща двойка цифри наляво. Предварително списъкът за резултата се инициализира с необходимата дължина. Цикълът за събирането се изпълнява толкова пъти, колкото са цифрите на по-дългото число. Когато цифрите на по-късото число „свършат”, се прибавят нули.

```
public static LongNum operator +(LongNum a, LongNum b)
{
    int a_len = a.Length, b_len = b.Length;
    int n = Math.Max(a_len, b_len);
    LongNum res = new LongNum("", n + 1);
    int t, c = 0;
    for (int i = 0; i < n; i++)
    {
        t = (i < a_len ? a.lnum[i] : 0) +
            (i < b_len ? b.lnum[i] : 0) + c;
        res.lnum.Add((byte)(t % 10));
        c = t / 10;
    }
    if (c == 1) res.lnum.Add(1);
    return res;
}
```

Реализирано е умножение на дълго число на едноцифрено десетично число. Цифрите започват да се умножават отдясно, като остатъкът до ***10*** се записва в съответната позиция, а преносът се добавя към следващата цифра получена цифра.

```
public static LongNum operator *(LongNum a, int x)
{
    int a_len = a.Length;
    LongNum res = new LongNum("", a_len + 1);
    int t, c = 0;
```

```
    for (int i = 0; i < a_len; i++)
    {
        t = a.lnum[i] * x + c;
        res.lnum.Add((byte)(t % 10));
        c = t / 10;
    }
    if (c > 0) res.lnum.Add((byte)c);
    return res;
}
```

Методът **LeftShift** служи за добавяне отдясно на ***k*** на брой десетични нули към число, което е равносилно на умножението на числото по ***10*** на степен ***k***.

```
private static LongNum LeftShift(LongNum x, int pos)
{
    LongNum res = new LongNum("", x.Length + pos);

    for (int i = 0; i < pos; i++) res.lnum.Add(0);
    res.lnum.AddRange(x.lnum);
    return res;
}

private LongNum GetRange(int pos, int len)
{
    LongNum res = new LongNum();
    res.lnum = this.lnum.GetRange(pos, len);
    return res;
}
```

За умножението на две дълги числа се използва следния алгоритъм: Трябва да умножим две числа ***a*** и ***b***. Разделяме цифрите на всяко едно от двете числа наполовина, или ***a = al * 10^k + ar , b = bl * 10^m + br*** , където ***k*** е броя на цифрите на дясната част на първото число, а ***m*** е броя на цифрите в дясната половина на ***b***. Резултатът от умножението е

$$a*b = al*bl*10^{k+m} + al*br*10^{k} + bl*ar*10^{m} + ar*br$$

Произведенията ***al * bl, al * br, bl * ar, ar * br*** получаваме с рекурсивно обръщение към същата операция умножение, умножението със степен на ***10*** получаваме с методът **LeftShift**.

```
public static LongNum operator *(LongNum a, LongNum b)
{
    if (a.Length == 1) return b * a.lnum[0];
    if (b.Length == 1) return a * b.lnum[0];

    int len_ra = a.Length - a.Length / 2;
    int len_rb = b.Length - b.Length / 2;

    LongNum la = a.GetRange(len_ra, a.Length / 2);
    LongNum ra = a.GetRange(0, len_ra);
    LongNum lb = b.GetRange(len_rb, b.Length / 2);
    LongNum rb = b.GetRange(0, len_rb);
```

```
            return LeftShift(la * lb, len_ra + len_rb)
                +            ra * rb
                +  LeftShift(la * rb, len_ra)
                +  LeftShift(ra * lb, len_rb);
        }

        public override string ToString()
        {
            char[] res = new char[this.Length];
            int i = this.Length-1;
            foreach (int el in this.lnum) res[i--] = (char)(el + '0');
            return new String(res);
        }
    }
```

Използването на класа **LongNum** може да стане например по следния начин:

```
    static void Main(string[] args)
    {
        Console.Write("a = ");
        LongNum a = new LongNum(Console.ReadLine());

        Console.Write("b = ");
        LongNum b = new LongNum(Console.ReadLine());

        LongNum c = a + b;
        Console.WriteLine(c);

        LongNum x = new LongNum(), y = new LongNum();

        x.Set("22222222222222222222222222222222222222222");
        y.Set("33333333333333333333333333333333333333333333333333");

        x *= y;
        Console.WriteLine(x);

        Console.ReadKey();
    }
```

По подобен начин лесно могат да се реализират изваждането на дълги цели числа и повдигане на дълго цяло число на степен цяло число.

3. Сортиране на масиви

В **C#** директно можем да използваме реализираната в **.NET** сортировка за масиви **System.Array.Sort**, в която се използва метода на бързото сортиране. В следващия пример сортираме масив, като използваме стандартния метод за сравнение на елементите.

```
using System;

namespace ConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int n;
            ...
            int[] my_array = new int[n];
            ...
            Array.Sort(my_array);
            ...
        }
    }
}
```

Освен това и различните класове от колекцията **Collections.Generic** имат методи **Sort()**.

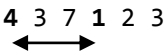
Ще разгледаме като примери някои от най-известните и основните методи за сортиране. В програмите ще сортираме масиви от цели числа в нарастващ ред, с индекси на масива от ***0*** до ***n-1***.

Тук ще декларираме и една функция, която ще използваме за размяна на елементи на масив:

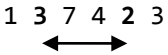
```
static class IntArraySwap
{
    static public void SwapElements(this int[] array,
                                     int indexOne, int indexTwo)
    {
        int tmp = array[indexOne];
        array[indexOne] = array[indexTwo];
        array[indexTwo] = tmp;
    }
}
```

3.1. Сортировка чрез пряка селекция (Selection sort)

На първата стъпка от сортировката намираме минималния елемент в масива и го разместваме с този на първо място.



На втората – намираме следващия минимален елемент от втория елемент нататък и го разместваме с този на второ място.



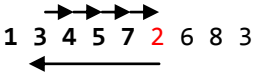
На стъпка ***i*** намираме минималния елемент в масива от позиция ***i-1*** до ***n-1*** и го разместваме с този на позиция ***i-1***.

Имаме ***n-1*** стъпки във външния цикъл. Във вътрешния цикъл, този за търсенето на минимален елемент, първо имаме ***n-1*** стъпки, след това ***n-2*** и така до като стигнем до ***1***. Общо имаме точно ***n.(n-1)/2*** стъпки, в които се сравняват елементи и имаме точно ***n-1*** размествания на елементи. Сложността на алгоритъма е $O(n^2)$. Можем да използваме тази сортировка само при масиви с малък брой елементи. Сортировката не е устойчива, т.е. равните елементи си променят относителната позиция. Това свойство на сортировките е важно, ако искаме да направим няколко сортировки на масиви със сложни елементи по няколко признака последователно.

```
static void SelectionSort(int[] arr)
{
    int i, j, pos;
    int n = arr.Length;
    int min_el;

    for (i=0; i<n-1; i++)
    {
        min_el = arr[i];
        pos = i;
        for (j=i+1; j<n; j++)
        {
            if (arr[j] < min_el)
            {
                min_el = arr[j];
                pos = j;
            }
        }
        arr.SwapElements(i, pos);
    }
}
```

3.2. Сортировка чрез вмъкване (Insertion sort)

Приемаме, че елементите на масива са сортирани до елемент с индекс ***k***. Присвояваме стойността на елемента ***k*** на променливата ***x***. Последователно сравняваме ***x*** с елементите вляво, ако ***x*** е по-малък от текущия елемент, то преместваме този елемент една позиция надясно. Ако ***x*** е по-голямо или равно, приключваме вмъкването, като поставим ***x*** на последното освободено място. На първата стъпка от външния цикъл (по позицията, до която е сортиран масива) правим едно сравнение и максимум едно преместване.



На втората – максимум две сравнения и премествания и така до последната с максимум ***n-1*** сравнения и премествания. Общо имаме максимум ***n.(n-1)/2*** премествания и сравнения. Порядъкът на операциите е ***O(n²)***. Намирането на позицията на елемента, който ще вмъкваме можем да направим и с двоично търсене в сортирания масив вляво, но пак ще трябва да преместим необходимите елементи, за да направим място на новия. Сортировката е устойчива. Може да се използва в случаите, когато не много често постъпват нови елементи, които трябва да се добавят в сортиран масив.

```
static void InsertionSort(int[] arr)
{
    int i, k;
    int x;
    int n = arr.Length;

    for (k = 1; k < n; k++)
    {
        x = arr[k];
        i = k - 1;
        while (i >= 0 && x < arr[i])
            arr[i + 1] = arr[i--];
        arr[i + 1] = x;
    }
}
```

3.3. Метод на мехурчето (Bubble Sort)

Идеята е следната: на първата стъпка започваме от лявата част на масива и сравняваме последователно всеки два съседни елемента ***i*** и ***i+1***, ако първият елемент е по-голям от вторият, ги разместваме. Когато сравним и последните два елемента, в дясно ще е „изплувал" най-големият елемент на масива. Изпълняваме същата процедура, но до предпоследния елемент и т.н. Отдясно изплуват следващите по големина елементи. Прекратяваме

сортировката, когато при поредната стъпка не сме разменили нито една двойка съседни елементи. Това означава, че масивът е сортиран. Порядъкът на броя на сравненията и разместванията е $O(n^2)$. Най-лошият вариант за тази сортировка е, когато първоначално масивът е сортиран точно обратно. Независимо че е една от бавните, можем да използваме метода на мехурчето, когато трябва да сортираме почти подредени масиви (с малък брой разместени елементи).

Можем малко да подобрим сортировката, като след стъпка със сравнения на съседни елементи надясно, направим стъпка със сравнения наляво, като наляво ще изплува следващ минимален елемент. По този начин елементите малко по-бързо отиват по местата си. Този метод се нарича *сортировка чрез клатене* (Shaker Sort).

```
static void BubbleSort(int[] arr)
{
    bool exchange;
    int i, k;
    int n = arr.Length;

    for (k = n - 1; k > 0; k--)
    {
        exchange = false;
        for (i = 0; i < k; i++)
            if (arr[i] > arr[i + 1])
            {
                arr.SwapElements(i, i + 1);
                exchange = true;
            }
        if (!exchange) break;
    }
}
```

3.4. Метод на Шел (Shell sort)

Това е един от най-бързите алгоритми със сложност $O(n^2)$. Подобрява метода на мехурчето и сортировката чрез вмъкване, като се сравняват несъседни елементи. Сложността на сортировката зависи от избора на стъпките за сравнение на елементи, като най-добрият резултат е $O(n.log\ n)$ или $O(n^{3/2})$. Не е устойчива сортировка.

Ще обясним идеята на сортировката със следния пример: Имаме масив с 16 елемента $a_1,\ldots,a_{16}$. Сортираме следните множества $\{a_1,a_9\}$, $\{a_2,a_{10}\}$, ... , $\{a_8,a_{16}\}$. Имаме стъпка между елементите равна на 8. След това сортираме $\{a_1,a_5,a_9,a_{13}\}$, $\{a_2,a_6,a_{10},a_{14}\}$, ..., $\{a_4,a_8,a_{12},a_{16}\}$ – стъпка 4. Сортираме $\{a_1,a_3,\ldots,a_{15}\}$ и $\{a_2,a_4,\ldots,a_{16}\}$ – стъпка 2. Накрая сортираме $\{a_1,a_2,\ldots,a_{16}\}$ – стъпка 1. Преди последното сортиране масивът е почти подреден и елементите няма да се преместят на повече от една позиция от местата си. Сортирането в реализацията се прави със сортиране чрез вмъкване. Този

метод още се нарича и *вмъкване с намаляваща стъпка*. В конкретната програма използваме в обратен ред стъпки ***1, 3, 7, 15, 31, ...*** . Първата стъпка зависи от дължината на масива. С тези стъпки сложността на програмата е $O(n^{3/2})$. Няма да разглеждаме различни редици от стъпки. В интернет може да се намери достатъчно информация за тях.

В конкретната програма в главния цикъл започваме от елемент с индекс равен на стъпката и обхождаме елементите до края. На всяка стъпка се опитваме да вмъкнем елемента в предходните, само че те не са последователни, а през стъпка **step.** Казано по друг начин вмъкваме елемента в списъка от предходни елементи, които са на разстояние **step**. Двата най-вътрешни цикъла са точно сортировката чрез вмъкване, но със стъпка по-голяма от ***1***. Най-външният цикъл е по избраните стъпки.

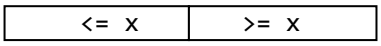
```
static void ShellSort(int[] arr)
{
    int step, i, k;
    int x;
    int n = arr.Length;

    step = 2; // използваме редицата от стъпки ... 31,15,7,3,1
    while (step < n) step <<= 1; // първата стъпка е по-малка от n
    step = step / 2 - 1;
     while (step >= 1)
    {
        for (i = step; i < n; i++)
        {
            x = arr[i];
            k = i - step;
            while (k >= 0 && arr[k] > x)
            {
                arr[k + step] = arr[k];
                k -= step;
            }
            arr[k + step] = x;
        }
        step = step / 2;
    }
}
```

3.5. Бърза сортировка (Quick sort)

Това е една от най-бързите сортировки и на практика се използва най-често, понеже е лесна за запомняне и реализация и няма сложни операции.

Алгоритъмът е следният: трябва да сортираме част от масив с дължина ***n*** и определена с ляв и десен край. Първо от подмасива си избираме някакъв елемент. Обикновено взимаме средния, но може да е първия, последния или произволен елемент.

Разделяме масива на две части спрямо стойността ***x*** на този елемент, така че в първата част елементите да са по-малки или равни на ***x***, а във втората – по-големи или равни на ***x***.



Това правим, като започнем с два индекса сочещи към елементите в началото и края на подмасива. Ако първият елемент е по-голям или равен от втория, ги разменяме. След това увеличаваме първия индекс с ***1***, а втория намаляваме с ***1***. Пак сравняваме двата елемента сочени от тези индекси и продължаваме по същия начин, докато индексите се разминат.

Броят на операциите в това разделяне на подмасива е равен на дължината му.

След като елементите в първата част са по-малки или равни на тези от втората и успеем да сортираме двете части на подмасива поотделно, ще получим сортиран подмасив. Така че се обръщаме рекурсивно към същата функция с лявата част на нашия подмасив и след това се обръщаме и с дясната част.

```
static void QuickSort(int[] arr, int lb, int rb)
{
    int i=lb, j=rb;
    int x;

    x = arr[(lb+rb)/2];
    while (i <= j) // разделяме масива на две части
    {
        while (i  <= rb && arr[i] < x) i++;
        while (lb <= j  && arr[j] > x) j--;
        if (i <= j)
        {
            arr.SwapElements(i, j);
            i++; j--;
        }
    }
    if (i < rb) QuickSort(arr, i, rb);
    if (lb < j) QuickSort(arr, lb, j);
}
```

Ако на всеки ход от сортировката разделяме подмасивите точно на две ще имаме ***log n*** на брой стъпки. За всяка една от тези стъпки трябва да подредим елементите по определения по-горе начин. Сумарно за всички подмасиви на една дълбочина на рекурсията операциите ще бъдат ***n*** на брой. Така че най-доброто постижение на тази сортировка е ***O(n log n)***. Ако на всеки ход подмасивите се разделят на примерно лява част от един елемент и дясна всички останали, броят на външните стъпки ще стане приблизително ***n***. В този случай сложността на сортировката ще стане $O(n^2)$. Това може да се случи, когато изходния масив е почти сортиран и изборът на елемент ***x*** е лош.

Скоростта на сортировката може да се подобри, като правим проверка, когато дължината на подмасива за сортиране стане по-малка от предварително избрана, не продължаваме с рекурсивно обръщение, а сортираме този подмасив чрез друг метод – мехурче или вмъкване.

3.6. Сортировка чрез сливане (Merge sort)

Идеята на тази сортировка е следната: Разделяме подмасива, който сортираме на две части, сортираме поотделно лявата и дясна част и след това ги сливаме. Тъй като разделяме подмасивите винаги на две, броят на външните стъпки или дълбочината на рекурсията ще бъде ***log n***. Броят на операциите при сливането на всяка дълбочина е пропорционална на ***n.*** Така че сложността на сортировката е ***O(n log n)***. Това обаче не прави тази сортировка по-бърза от предходната, понеже при сливането на двете сортирани части на подмасив се налага преместването на поне едната част в друг масив. Сливането в нашия пример е направено по този начин:

```
static int[] tmp_arr;

static void MergeArr(int[] arr, int left_i, int middle_i,
                     int right_i)
{
    int i, j, k;

//  Тук преместване първата част от нашия масив във временен

    i = 0; j = left_i;
    while (j <= middle_i) tmp_arr[i++] = arr[j++];

// следва самото сливане на масивите, като резултата се записва в
// началото на масива

    i = 0; j = middle_i + 1; k = left_i;
    while (k < j && j <= right_i)
        if (tmp_arr[i] <= arr[j])
            arr[k++] = tmp_arr[i++];
        else
            arr[k++] = arr[j++];

// добавяме останалите елементи от помощния масив

    while (k < j) arr[k++] = tmp_arr[i++];
}

static void MergeSort(int[] arr, int left_i, int right_i)
{
    int middle_i;
    if (left_i < right_i)
    {
        middle_i = (left_i + right_i) / 2;
        MergeSort(arr, left_i, middle_i);
        MergeSort(arr, middle_i + 1, right_i);
        MergeArr(arr, left_i, middle_i, right_i);
    }
}
```

3.7. Пирамидална сортировка (Heap sort)

Наричаме пирамида (heap) двоично дърво със следното свойство. Елементите на двоичното дърво са подредени така, че а) стойността във върха е по-голяма (по-малка) от тези на наследниците; б) стойността във върха е по-малка (по-голяма) от тази в предходника на върха.

Реално в задачата работим вместо с двоично дърво с едномерен масив. Съответствието между елементите на графа и дървото, ако започваме с индекс ***0***, е следното: елемент от масива с индекс ***v*** има ляв наследник с индекс ***v*2+1**, десен наследник ***v*2+2*** и предходен елемент с индекс ***v/2-1***.

Първо построяваме пирамида, която е с най-голям елемент на върха.

След това на ***n-1*** на брой стъпки изпълняваме следната процедура: Разменяме върха на пирамидата с последния елемент. Последният елемент става най-големия и след това го изключваме от пирамидата. Пренареждаме новия елемент от върха, така че дървото пак да бъде пирамида, т.е. на върха ще дойде следващият по големина елемент. Пак разменяме елемента на върха и последния елемент в пирамидата и продължаваме така, докато пирамидата остане с един елемент. Изключените от пирамидата елементи са сортирани в нарастващ ред.

Сега по-подробно ще обясним построяването на пирамидата. Започваме с празна пирамида и последователно добавяме към нея елементите от масива за сортиране. За всеки нов елемент извършваме следното действие: ако е по-голям от предходния му елемент, ги разместваме и след това продължаваме проверката с неговия предходен елемент. Така, ако новият елемент е максималният, то той ще се изкачи до върха на пирамидата. Накрая, тъй като всеки елемент в дървото е по-малък от предходника му, то имаме пирамида.

```
static void BuildHeap(int[] arr)
{
    int n = arr.Length;
    int p, v;
    for (int k = 1; k < n; k++)
    {
        v = k;
        p = (v-1)/2;
        while (p < v && arr[p] < arr[v])
        {
            arr.Swapelements(p, v);
            v = p;
            p = (v-1)/2;
        }
    }
}
```

Понеже височината на пирамидата е най-много ***log n***, за да намерим мястото на новопостъпилия елемент, имаме най-много ***log n*** сравнения с предходници. Имаме ***n*** на брой елемента, така че операциите са сравними с ***O(n log n)***.

Сега разглеждаме процедурата по пренареждането на върха на пирамидата. Намираме по-големия от двата наследника от върха. Ако стойността на избрания наследник е по-голяма от тази на върха, то разменяме върха с по-големия наследник. След това правим същото с наследника и проверяваме за по-големия от неговите наследници. Продължаваме, докато текущият връх стане по-голям и от двата негови наследника.

```
static void HeapDown(int[] arr, int len)
{
    int ch = 1;
    int v = 0;
    while (ch < len)
    {
        if (ch + 1 < len && arr[ch] < arr[ch+1])
            ch++;
        if (arr[v] >= arr[ch]) break;
        arr.SwapElements(v, ch);
        v = ch;
        ch = v*2 + 1;
    }
}
```

Тази процедура зависи от височината на пирамидата. На всеки ход имаме най-много по две сравнения и разместване, така че максималния брой операции ще бъде ***3 log n***. Понеже извършваме тази операция ***n-1*** пъти, след размяна на върха на пирамидата и последния и елемент, общият брой операции ще е максимум ***3n log n.*** Заедно с операциите от построяването на пирамидата стават ***4n log n***. Общият порядък е най-много ***O(n log n).*** Това е по-добър резултат от този на *бързата сортировка*, обаче константата ***4*** пред броя на операциите реално тежи и освен това реализацията е по-тежка, има допълнителни проверки за последен елемент и т.н. На практика тази сортировка е по-бавна от *бързата сортировка* и почти не се използва.

```
static void HeapSort(int[] arr)
{
    BuildHeap(arr);

    int n = arr.Length;
    for (int k = n - 1; k > 0; k--)
    {
        arr.SwapElements(0, k);
        HeapDown(arr, k);
    }
}
```

3.8. Броене на честоти (Counting sort)

Идеята на тази сортировка е следната: Обхождаме един път елементите на масива, който ще сортираме и за всеки елемент „отбелязваме" колко пъти се среща в масив. Преброяването може да направим като използваме статичен масив с дължина максималната стойност на типа на масива, който сортираме или два пъти по-голяма, ако в масива има и отрицателни числа. Това ограничава типа на изходния масив до **byte** или **short**. Не можем да декларираме масив с дължина максималната стойност за тип **int**. Всеки пореден елемент на работния масив с индекс ***k*** показва колко пъти се среща елемента ***k*** в изходния масив. Сортировката се изпълнява с линейна скорост, но сме ограничени в стойностите на елементите.

```
static void CountingSort(short[] arr)
{
    int[] counts = new int[short.MaxValue];

    for (int i = 0; i < arr.Length; i++)
        counts[arr[i]]++;

    for (int i = 0, k = 0; i < counts.Length; i++)
        for (int j = 0; j < counts[i]; j++)
            arr[k++] = (short)i;
}
```

Друг възможен начин да преброим колко пъти се срещат елементите е да използваме класа **System.Collections.Generic.SortedDictionary,** който поддържа динамичен подреден списък от двойки <***key***, ***value***>. В ***key*** ще записваме стойностите на елементите от изходния масив, а ***value*** ще показва колко пъти се среща съответният елемент в изходния масив.

За всеки елемент от изходния масив се прави търсене в сортирания речник от двойки елементи (двоично търсене) и общата сложност на сортировката ще стане по-малка от ***O(n.log n).*** Сложността не е линейна, но изходният масив може да е от тип **int** или **long.**

```
static void CountingSort(int[] arr)
{
    SortedDictionary<int, int> counts
        = new SortedDictionary<int, int>();

    for (int i = 0; i < arr.Length; i++)
        if (counts.ContainsKey(arr[i]))
            counts[arr[i]]++;
        else
            counts.Add(arr[i],1);
    int k=0;

    foreach(KeyValuePair<int, int> kvp in counts)
        for (int i = 0; i < kvp.Value; i++)
            arr[k++] = kvp.Key;
}
```

4. Обхождане с връщане назад

Обхождането (търсене) с връщане назад (*Backtracking*) е метод, при който решението на задачата се конструира последователно. На всяка стъпка се прави опит да се разшири частичното решение, така че да отговаря на условието на задачата. Ако успеем, продължаваме нататък, а като се изчерпят всички възможности за разширяване, то това няма да доведе до решение на задачата и се връщаме на предходна стъпка.

Ще разгледаме няколко задачи, с чиято помощ ще изясним метода.

Генериране на низ без повтарящи се съседни поднизове

С три символа да се конструира низ с дължина ***100***, в който да няма повтарящи се съседни поднизове.

Приемаме, че сме конструирали низ с дължина ***i-1***.

- Опитваме да поставим първия символ на позиция ***i***;
- Ако не отговаря на условието, опитваме със следващ символ;
- Ако намерим символ, отговарящ на условието, преминаваме на позиция ***i+1*** и започваме отначало;
- Ако не намерим символ за тази позиция, се връщаме на предходната и проверяваме дали можем да поставим на тази позиция следващ символ;
- Продължаваме по този начин или до като ***i*** стане равно на ***100*** или ***i*** стане по-малко от ***0***, което означава, че нямаме решение.

За проверката за съществуване на повтарящи се съседни поднизове при нов поставен символ използваме функцията **check_substrings**.

```
static char[] w_str;

static bool check_substrings(int pos)
{
    int i, j;
    bool flag;

    for (i = 1; i <= pos/2; i++)
    {
        flag = true;
        for (j = pos; j > pos-i; j--)
            if (w_str[j] != w_str[j-i])
            {
                flag = false;
                break;
            }
       if (flag) return true;
    }
    return false;
}
```

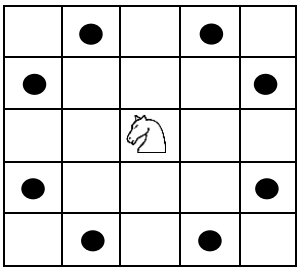
```
static void MakeString()
{
    int n = 100;
    w_str = new char[n];
    int i;

    w_str[0] = w_str[1] = 'a';
    i = 1;
    while (i >= 0 && i < n)
    {
        if (!check_substrings(i))
        {
            // няма съседни повтарящи се поднизове
            // минаваме на следваща позиция и поставяме първия символ
            i++;
            if (i < n) w_str[i] = 'a';
        }
        else
        {
            // има повтарящи се съседни поднизове
            // опитваме със следващия символ
            w_str[i]++;

            while (i >= 0 && w_str[i] > 'c')
            {
                // ако няма следващ възможен символ
                // се връщаме на предходна позиция
                i--;
                if (i >= 0) w_str[i]++;
            }
        }
    }

    if (i == n)
        Console.WriteLine(w_str);
    else
        Console.WriteLine("Няма решение");
}
```

Разходката на коня

Нека е дадена шахматна дъска. На дъската е разположена фигурата кон. Играчът може да движи коня според правилата на шаха. Целта на задачата е коня да обиколи всички позиции на дъската, така че всяка една от тях да бъде посетена точно веднъж.

Приемаме, че дъската е представена като двумерен масив с размер N x N, а началните координати на коня са дадени като целочислени променливи X и Y. За обхождането дефинираме помощни масиви **offsetX** и **offsetY** с допустимите движения на фигурата. Дефинираме помощен масив **visited**, съдържащ информация за посетените позиции на дъската.

- Маркираме обхождaната позиция за посетена;
- Ако всички позиции на дъската са вече посетени, значи сме намерили решение и връщаме положителен резултат;
- Последователно се опитваме да поставим коня на всички достижими позиции от мястото, където той се намира в момента;
- Ако позицията не е обходена и се намира в рамките на дъската я обхождаме;
- Ако обхождането на позицията върне положителен резултат, значи е намерено решение;
- Ако не, маркираме обхождaната позиция като непосетена и връщаме отрицателен резултат.



```
private int[] offsetX = new int[] { -2, -2,  1, -1,  2,  2,  1, -1 };
private int[] offsetY = new int[] {  1, -1,  2,  2,  1, -1, -2, -2 };

private int[ , ] visited;

public readonly int BoardSize = 8;

private bool AllNodesVisited()
{
    for (int i = 0; i < this.BoardSize; i++)
    {
        for (int j = 0; j < this.BoardSize; j++)
        {
            if (visited[i, j] <= 0)
                return false;
        }
    }
    return true;
}

public bool Solve(int x, int y, int current)
{
    visited[x, y] = current;

    if (AllNodesVisited()) return true;
```

```
    for (int i = 0; i < offsetX.Length; i++)
    {
        int newX = x + offsetX[i];
        int newY = y + offsetY[i];

        if (((newX > -1) && (newX < this.BoardSize)) &&
           ((newY > -1) && (newY < this.BoardSize)) &&
           (visited[newX, newY] <= 0))
        {
            if (Solve(newX, newY, current + 1))
                return true;
        }
    }

    visited[x, y] = 0;

    return false;
}
```

Симетрични квадрати от думи

Пъзелът „квадрат от думи" е специален вид акростих. Той се състои от набор от думи, написани в квадратна мрежа, така че едни и същи думи могат да се четат както хоризонтално, така и вертикално. Броят на думите, който е равен на броя на буквите във всяка една дума, се нарича *ред на квадрата*. Примери за квадрати от ред ***4*** и ***5***:

```
B E A R       C H I P       A P P L E       C R E S T
E L S E       H E R O       P L A I N       R E A C H
A S I A       I R O N       P A I N T       E A G E R
R E A L       P O N D       L I N E R       S C E N E
                            E N T R Y       T H R E E
```

Задачата е да генерираме всички възможни квадрати от даден ред, като използваме предварително зададен речник от думи.

За да запазим думите от речника в паметта на компютъра използваме списък **word_list** от тип **List<string>.** Речникът, който е от думи с еднаква дължина, се прочита от файл и се запазва в списъка. След това списъкът се сортира, като се използва директно методът **Sort()** на класа **List**.

```
public class WordSquareGenerator
{
    int sq_order = 5;
    char[ , ] w_square;
    List<string> word_list;
```

Когато се опитваме да намерим подходяща дума за следващия ред и стълб на квадрата, трябва да намерим всички думи в речника, започващи с определен низ. Това е реализирано със следващата функция. При

обръщение към функцията с **index** равен на ***-1***, търсим първата дума започваща с низа **c1**. Използваме метода на класа **List** за двоично търсене, който или намира дума и връща индекса и в списъка, или връща позицията, където би трябвало да е думата, умножена по ***-1***. Взимаме абсолютната стойност на резултата и проверяваме дали думата на тази позиция започва с търсения подниз. След това, когато ни трябва следваща дума започваща с този подниз, само се увеличава индекса с ***1*** и се проверява следващата дума. Когато не намерим първа дума или думите започващи със **c1** се изчерпят, връщаме стойност ***-1***.

```
int FindNextWord(string c1, int index)
{
    if (index == -1) // първо търсене на дума
    {
        index = word_list.BinarySearch(c1);
        if (index > 0) return index;
        index = Math.Abs(index) - 1;
    }
    else
        index++;

    if (index < word_list.Count &&
        word_list[index].Substring(0, c1.Length) == c1) return index;
    return -1;
}
```

Следващите две функции използваме за добавяне на нова дума в квадрата на ред и стълб ***r*** и за взимане на първите r няколко букви стълб ***r*** от квадрата, които ни трябват, за да намерим подходяща дума за това място.

```
void PutWordInSquare(string word, int r)
{
    int i = 0;
    foreach (char ch in word) w_square[r, i++] = ch;
}

string GetWordAtColumn(int r)
{
    string result = "";
    for (int i=0; i<r; i++) result += w_square[i,r];
    return result;
}

public WordSquareGenerator( )
{
    // четем речника от файл и го записваме в списъка от думи
    word_list =
        System.IO.File.ReadAllLines("dict.txt").ToList<string>();

    word_list.Sort();
    w_square = new char[sq_order, sq_order];
```

За да запомним за всеки ред от квадрата до коя дума сме стигнали и низа, с който търсим думите на този ред, използваме два помощни масива **w_pos** и **search_str**.

```
List<string> square_list = new List<string>();
string[] search_str = new string[sq_order];
int[] w_pos = new int[sq_order];
int row = 0;
```

За първата дума в квадрата използваме отделен цикъл, в който поставяме последователно думите от речника и след това започваме да проверяваме дали можем да поставим подходяща дума на следващия ред и стълб. Проверката за ред с номер row става по следния начин:

- Ако row е равен на реда на квадрата, сме намерили поредното решение, записваме редовете на квадрата в списъка **square_list** и намаляваме ***row*** с ***1***;
- Ако ***row*** е по-малко от реда на квадрата, търсим следваща дума, започваща с първите row символа от ред ***row***;
- Ако имаме следваща дума, записваме я в квадрата и увеличаваме ***row*** с ***1***;
- Ако няма следваща дума за този ред и стълб, намаляваме ***row*** с ***1***. При връщането на предходния ред в **w_pos[row]** е запазена позицията на думата на този ред, а в **search_str[row]** е записан низа, с които търсим дума, така че като се върнем отново в началото на цикъла, ще потърсим следваща подходяща дума;
- Цикълът завършва, когато ***row*** стане равно на нула, т.е. сме изчерпали всички възможности за квадрат с текущата първа дума.

Това е и реализация на променлив брой вложени един в друг цикли.

```
w_pos[0] = -1;

foreach (string word in word_list)
{
    PutWordInSquare(word, 0);

    w_pos[0]++;
    row = 1; w_pos[row] = -1;

    while (row > 0)
    {
        if (row < sq_order)
        {
            if (w_pos[row] == -1)
                search_str[row] = GetWordAtColumn(row);

            w_pos[row] = FindNextWord(search_str[row], w_pos[row]);
```

```
                    if (w_pos[row] >= 0)
                    {
                        PutWordInSquare(word_list[w_pos[row]], row);
                        row++;
                        if (row < sq_order) w_pos[row] = -1;
                    }
                    else
                        row--;
                }
                else
                {
                    for (int i=0; i < sq_order; i++)
                        square_list.Add(word_list[w_pos[i]]);

                    square_list.Add("");
                    row--;
                }
            }
        }
        System.IO.File.WriteAllLines("w_square.txt",
                                     square_list.ToArray<string>());
    }
}
```

За повечето задачи от този вид това е единственото приемливо възможно решение. Ако в предходните задачи се опитаме да генерираме последователно всички възможни варианти и за всеки от тях да проверим дали отговаря на условията на задачата, то това няма да стане в реално време. Например, при задачата за дамите на шахматна дъска, които не трябва да се бият, всички възможни варианти с неповтарящи се позиции на дамите са 8^8.

5. Генериране на комбинаторни обекти

Тук ще разгледаме генерирането на пермутации и комбинации, които често се използват в задачи, в които се налага размяна на елементи или получаване на различни варианти, за които се проверява дали са решение на задачата.

Получаване на пермутации с рекурсия

Използваме следната схема за получаване на пермутации, например за ***5*** елемента намираме пермутациите:

1, пермутации (2, 3, 4, 5)
2, пермутации (1, 3, 4, 5)
3, пермутации (1, 2, 4, 5)
4, пермутации (1, 2, 3, 5)
5, пермутации (1, 2, 3, 4)

т.е. от числата, с които ще генерираме комбинации, взимаме всяко число, поставяме го на първо място и с останалите елементи се обръщаме рекурсивно към функцията.

Ще работим с едномерен масив и с числата от ***1*** до ***n,*** където ***n*** е броя на елементите, с които ще генерираме пермутации. В програмата обръщението към рекурсивната функция ще става по следния начин: на дълбочина ***k*** от обръщението имаме подредени ***k*** елемента и се опитваме да получим пермутации от останалите, например за ***n=5*** и ***k=2*** имаме:

2 4 1 пермутации([3, 5], k+1)
2 4 3 пермутации([1, 5], k+1)
2 4 5 пермутации([1, 3], k+1)

Трябва да опишем, как ще получаваме по едно поредно число от останалите след позиция ***k***. Първото число ще е на мястото си за първото обръщение. Правим цикъл за останалите и на първата стъпка от този цикъл разменяме първото число от останалите, което е на позиция ***k+1*** с това на позиция ***k***, на втората заменяме числата на позиции ***k*** и ***k+2***, на стъпка ***i*** заменяме числата на позиция ***k*** и ***k+i***.

Има още нещо, за което трябва да помислим. След излизането от рекурсията, числата след позиция ***k*** ще бъдат обърнати в обратен ред. Трябва да обърнем симетрично тези числа и след това да заменяме с елемента на място ***k.*** Предходния пример ще изглежда така:

2 4 1 пермутации([3, 5], k+1)

след обръщението в масива ще имаме

2 4 1 **5 3** -> **2 4** 1 **3 5**

обръщаме последните два елемента в масива и след това разменяме ***1*** и ***3***

2 4 3 пермутации([1, 5], k+1)
2 4 3 **5 1** -> **2 4** 3 **1 5**

разменяме ***3*** и ***5***

2 4 5 пермутации([1, 3], k+1)

С функцията **swap_perm** обръщаме симетрично подмасив от позиция ***i*** до позиция ***j***.

```
static int[] perm;

static void swap_perm(int i, int j)
{
    while (i < j) perm.SwapElements(i++, j--);
}

static void Permutation(int pos, int n)
{
    int i;

    if (pos == n) // има намерена пермутация и я отпечатваме
    {
        for (i = 0; i < n; i++) Console.Write(perm[i]);
        Console.WriteLine();
        return;
    }
    Permutation(pos + 1, n);
    for (i = 1; i < n-pos; i++)
    {
        swap_perm(pos + 1, n - 1);
        perm.SwapElements(pos, pos + i);
        Permutation(pos + 1, n);
    }
}

static void StartPerm()
{
    int n;
    Console.Write("n="); n = int.Parse(Console.ReadLine());
    perm = new int[n];
    for (int i = 0; i < n; i++) perm[i] = i + 1;
    Permutation(0, n);
}
```

Последователно генериране на пермутации

Понякога се налага да получаваме пермутациите последователно и в лексикографски порядък, т.е. трябва да намерим алгоритъм как от една пермутация да получим следващата. Това не можем да направим с рекурсивната функция.

Ако последната цифра на пермутацията е по-голяма от предпоследната, ги разменяме и получаваме следващата пермутация

`1 2 3 4 5 6` -> `1 2 3 4 6 5`

В другия случай намираме максимално дълга редица от нарастващи елементи отзад напред и това е последната възможна пермутация с тази редица от елементи, това отговаря на положението след рекурсивното обръщение към функцията в предходната задача. Приемаме, че тази редица започва от позиция ***k+1***. Обръщаме елементите в масива от индекс ***k+1*** до ***n-1.***

`1 4 3 6 5 2` -> `1 4 3 2 5 6` – не е следваща пермутация

След това намираме надясно първото число по-голямо от това на позиция ***k*** и заменяме двете числа. Това пак отговаря на рекурсивния вариант, само че в случая не знаем веднага, кое число да заменим с това от позиция ***k***. Получаваме `1 4 5 2 3 6`.

```
static bool NextPerm(int[] arr)
{
    int i, k, left, right;

    k = arr.Length-1;
    while (k > 0 && arr[k-1] > arr[k]) k--;
    if (k < 1) return false;

    left = k; right = arr.Length-1;
    while (left < right)
        arr.SwapElements(left++, right--);

    i = k--;
    while (arr[k] > arr[i]) i++;
    arr.SwapElements(k, i);

    return true;
}
```

А така можем да се обърнем към функцията **`NextPerm`**

```
perm = new int[4];
for (int i = 0; i < perm.Length; i++) perm[i] = i + 1;
do
{
    foreach (int p in perm) Console.Write(p);
    Console.WriteLine();
}
while (NextPerm(perm));
```

Генериране на комбинации

Разглеждаме получаването на комбинации от n елемента от k-ти клас. Това означава, че всяка комбинация се състои от k от зададените n елемента, като редът на елементите е без значение. Пример за $C(5,3)$:

```
1 2 3       1 3 5       2 3 5
1 2 4       1 4 5       2 4 5
1 3 4       2 3 4       3 4 5
```

Получаваме комбинациите итеративно, като методът е малко опростено обхождане с връщане назад. От текущата позиция i в цикъла, запълваме до края на масива с нарастващи елементи, като последният не трябва да става по-голям от n. Отпечатваме комбинацията. След това се намираме първия елемент наляво, който е по-малък от $n-k+i$. Смисълът е, че на всяка позиция от масива имаме максимална възможна стойност и когато увеличим елемента с 1, той не трябва да надхвърли тази стойност.

```
static void Combinations()
{
    int n, k;
    Console.Write("n="); n = int.Parse(Console.ReadLine());
    Console.Write("k="); k = int.Parse(Console.ReadLine());
    int [] comb = new int [k];

    int i=0;
    comb[i] = 1;

    while (i >= 0)
    {
        while(i < k-1)
        {
            i++;
            comb[i] = comb[i-1]+1;
        }
        for (int c = 0; c < k; c++) Console.Write(comb[c]);
        Console.WriteLine();

        comb[i]++;

        while (i >= 0 && comb[i] > n-k+i+1)
        {
            i--;
            if (i >= 0) comb[i]++;
        }
    }
}
```

6. Двоични дървета и графи

6.1. Представяне на дървета и графи

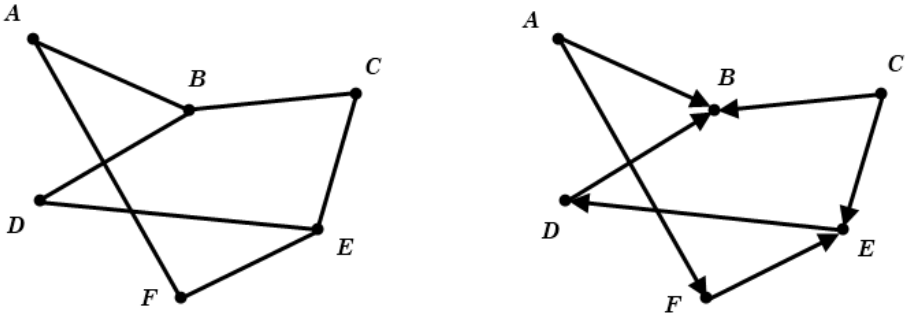
В стандартните библиотеки на C# няма реализирани класове за представяне на графи и дървета. Програмистът сам трябва да декларира тези класове или структури. Могат да се намерят различни реализации за двоични дървета и графи, освен това в MSDN има подробно описание на динамичните структури от данни и как да бъдат реализирани със средствата на C#. В зависимост от конкретната задача може да бъде избрано различно представяне.

Най-общо можем да представим връх (елемент) на произволно дърво, като информационна част и списък от наследници. Списъкът може да е от номера на върховете наследници, а може и да е от декларирания тип (клас) на върха. Елементът може и да съдържа инстанция или номер на неговия предходник.

За обхождане на върховете на дървото, започваме от дадан връх и след това рекурсивно обхождаме наследниците му.

Елементите на двоичните дървета обикновено инстанции от съответния клас за ляв и десен наследник.

Графът е множество от върхове (nodes) и свързващите ги ребра (edges).



В зависимост от конкретните задачи можем да използваме различни представяния на графите. Например:

- Списък от двойки върхове (ребра), като двойките може да са наредени или не в зависимост от това дали графа е ориентиран или не. Например, двата графа от горната картинка ще се представят, като: ***{{A,B}, {A,F}, {B,C}, {B,D}, {C,E}, {E,F}, {D,E}}*** и ориентирания граф: ***{{A,B}, {A,F}, {C,B}, {C,E}, {D,B}, {E,D}, {F,E}}***.
- Матрица на съседства. Размерът и е ***n* x *n*,** където ***n*** е броят на върховете на графа. Елемент на матрицата с индекси ***i*** и ***j*** представя реброто от връх ***i*** до връх ***j*.** Можем да се интересуваме

само от това дали има връзка между върховете и тогава стойностите на елементите ще бъдат ***0*** или ***1***. Елементите могат да съдържат теглото (напр. дължина) на реброто. Ако графът е неориентиран матрицата ще бъде симетрична относно главния диагонал. При ориентиран граф, стойността на елемента ***i, j*** за ребро от ***j*** към ***i*** може да бъде 0 или стойност на реброто. Матриците за съседства за горните два графа ще са:

	A	*B*	*C*	*D*	*E*	*F*
A	0	1	0	0	0	1
B	1	0	1	1	0	0
C	0	1	0	0	1	0
D	0	1	0	0	1	0
E	0	0	1	1	0	1
F	1	0	0	0	1	0

	A	*B*	*C*	*D*	*E*	*F*
A	0	1	0	0	0	1
B	0	0	0	0	0	0
C	0	1	0	0	1	0
D	0	1	0	0	0	0
E	0	0	0	1	0	0
F	0	0	0	0	1	0

- Във всеки елемент на дървото ще се съдържа списък с наследниците на върха. Елементите на списъка са най-общо указатели към наследниците или индекси на тези върхове в някаква поддържаща структура. Например, за първият граф ще имаме:

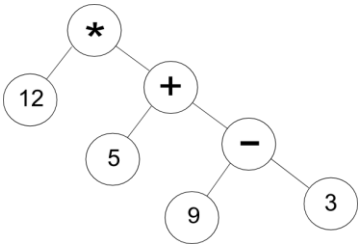
  A – { B, F }
  B – { A, C, D }
  C – { B, E }
  D – { B, E }
  E – { C, D, F }
  F – { A, E }

Всяко едно от представянията си има различни предимства и недостатъци.

Ако повдигнем матрицата съседства на степен ***k***, в елементите на резултата ще означават: ненулевата стойност ще означава, че има път с дължина ***k*** между съответните върхове. Този резултат е верен и за ориентирани и неориентирани графи. Ако стойността на елемент ***i, j*** е броят на ребрата между двата върха, като това се отнася и за ребро от връх към себе си, то положителната стойност на резултата ще показва още и броя на всички различни пътища с дължина ***k*** между два върха. Това представяне не е удобно, когато броят на ребрата е малък, т.е. матрицата е разредена.

6.2. Обхождане на двоични дървета

Имаме аритметичен израз записан в двоично дърво по следния начин: Във всеки връх, които има два наследника, има записана двуаргументна операция, а във върховете листа, т.е. тези които нямат наследник има записана стойност или променлива. Задачата е да се напише програма, която изчислява израза, записан в двоичното дърво. За да можем да тестваме програмата, ще инициализираме дървото с израз, записан в обратен полски запис. Например стойността от дървото на картинката ще бъде ***12 * (5 + 9 - 3) = 132.***



Ще използваме собствени класове за описание на двоичното дърво и на елементите на дървото.

Задачата ще решим рекурсивно, като за всеки връх, в които имаме операция, намираме стойността на лявото поддърво, стойността на дясното поддърво и връщаме като резултат операцията изпълнена върху двете стойности. Ако върхът е листо, връщаме стойността записана вътре. За опростяване на реализацията, няма да използваме променливи.

```
public class BinaryTreeNode
{
    private double _value;
    private  char  _oper;
    private BinaryTreeNode _left;
    private BinaryTreeNode _right;

    public BinaryTreeNode(double value)
    {
        _value = value;
        _oper = ' ';
        _left = _right = null;
    }

    public BinaryTreeNode(char op)
    {
        _oper = op;
        _left = _right = null;
    }

    public double Calc()
    {
        if (_oper == ' ') return _value;
```

```
            switch (_oper)
            {
                case '*':
                    return _left.Calc() * _right.Calc();
                case '/':
                    return _left.Calc() / _right.Calc();
                case '+':
                    return _left.Calc() + _right.Calc();
                case '-':
                    return _left.Calc() - _right.Calc();
            }
            return _value;
        }
    }

    public class ExprTree
    {
        private BinaryTreeNode _root;
        private double _value;

        public ExprTree()
        {
            _root = null;
        }
        public double CalcTree()
        {
            _value = _root.Calc();
            return _value;
        }
    }
```

След декларация и записване на израз в дървото, можем да пресметнем израза по следния начин:

```
ExprTree expr1 = new ExprTree();

...

double result = expr1.CalcTree();

...
```

За да инициализираме двоичното дърво, можем да използваме зададен израз в обратен полски запис. Следващата функция има за вход израз в постфиксен запис и го записва в двоичното дърво. В случая елементите на израза са едноцифрени положителни числа и операциите **+, -, *** и /. Алгоритъмът е почти същия като пресмятането на израз в постфиксен запис. Когато пристигне операция от стека се изваждат двата върха и се присвояват като ляв и десен наследник на новия връх, който съдържа операция. Новият връх се добавя в стека.

```
public BinaryTreeNode PolishNotation2Tree(string pf_equation)
{
    BinaryTreeNode new_node;
    Stack<BinaryTreeNode> working_stack = new Stack<BinaryTreeNode>();

    foreach (char ch in pf_equation)
    {
        switch (ch)
        {
            case '*':
            case '/':
            case '+':
            case '-':
                new_node = new BinaryTreeNode(ch);
                new_node.Right = working_stack.Pop();
                new_node.Left  = working_stack.Pop();
                working_stack.Push(new_node);
                break;
            default:
                if (char.IsDigit(ch))
                {
                    new_node = new BinaryTreeNode(ch - '0');
                    working_stack.Push(new_node);
                }
                break;
        }
    }
    return working_stack.Pop();
}
```

6.3. Обхождане на графи

Ще разгледаме следната задача: За две зададени думи с еднаква дължина да се намери редица от думи (с минимална дължина), с която след преобразуване на една дума в следваща дума (съседни думи), получаваме от първата зададена дума – втората. Преобразуването на една дума в друга става с промяна на само една буква. За валидни думи използваме предварително зададен речник с думи с еднаква дължина, който е сортиран лексикографски. Редицата думи от първата до последната се нарича *път*.

Например, с речника, с който разполагаме получаваме следния минимален път между думите **math** и **rose**:

math – mate – male – mole – role – rose

Трябва да можем да получаваме една дума от друга. Вариантът, в който генерираме всички възможни низове и проверяваме всеки от тях, дали се съдържа в речника, е неприемлив, тъй като възможните съседни низове, например за четирибуквени английски думи е 26^4.

Така че след зареждането на речника в паметта ще намерим всички съседни думи в речника и ще получим неориентиран граф с върхове от думи и ребра, свързващи съседни думи.

Ще решим задачата по два начина, като използваме различно обхождане на получения граф.

Търсене в ширина (Breadth-first search)

Обхождането на граф в ширина е начин да се обходят всички върхове в граф и се изразява в следното: Взимаме един връх от графа, намираме всички наследници на върха и ги обработваме. Следващият връх, с който извършваме същите действия, взимаме от най-рано обработените наследници. Ако няма обработени наследници, взимаме друг необработен връх. Продължаваме, докато обработим (обходим) всички върхове или докато намерим решението, което търсим.

Реализацията става така: Започваме с даден връх от графа, маркираме го и го добавяме в опашка. След това докато опашката не е празна извършваме следните действия. Взимаме първия връх от опашката и проверяваме дали думата от върха е тази, която търсим. След това намираме всички немаркирани наследници на този връх, маркираме ги и ги добавяме към опашката. Ако опашката е празна, трябва да проверим дали няма други необработени думи в графа.

Понеже трябва да намерим и конкретния път между двете думи, когато взимаме следващ връх от опашката, не го премахваме от опашката.

Опашката съдържа номер на връх, дълбочината на обработка на върха и номера на предходника на върха от опашката.

```
public class WordPathFinder
{
    class QueueElement
    {
        public int v { get; set; }
        public int d { get; set; }
        public int p { get; set; }
    }
    Queue<QueueElement> my_queue;
    QueueElement q;
```

За представяне на графа от думи използваме едномерен масив от списъци, като в списък с пореден номер ***k*** записваме последователните номера на думите съседни с думата с номер ***k.*** Думите от речника са записани в масив от низове.

```
    string[] dict;
    List<string> w_res;
    Boolean[] mark;
    List<int>[] desc;
```

Използваме следващата функция, за да проверим дали две думи са съседни, т.е. дали се различават само по една буква.

```
bool str_neigh(string a, string b)
{
    int n = a.Length;
    int i, diff = 0;

    for (i = 0; i < n; i++)
    {
        if (a[i] != b[i]) diff++;
        if (diff > 1) return false;
    }
    if (diff == 1) return true;
    return false;
}
```

С функцията **DisplayPath** намираме пътя между думите в обратен ред.

```
void DisplayPath()
{
    int k = my_queue.Count-1;
    while (k > -1)
    {
        q = my_queue.ElementAt(k);
        k = q.p;
        Console.WriteLine(dict[q.v]);
    }
}
```

Понеже масивите нямат метод за двоично търсене сме реализирали двоично търсене, което се използва, за да се намерят позициите на двете зададени думи.

```
int FindWord(string str)
{
    int l, r, m, cmp;
    l = 0; r = dict.Length - 1;

    while (l <= r)
    {
        m = (l + r) / 2;
        cmp = str.CompareTo(dict[m]);
        if (cmp == 0)
            return m;
        else if (cmp < 0)
            r = m - 1;
        else
            l = m + 1;
    }
    return -1;
}
```

```
bool SearchWordsPath(string w1, string w2)
{
    int k;
    k = FindWord(w1);
    if (k == -1) return false;

    mark[k] = true;
    my_queue.Enqueue(new QueueElement() { v = k, d = 1, p = -1 });

    k = FindWord(w2);
    if (k == -1) return false;

    // основен цикъл за обхождането в ширина
    k = 0;
    while (k < my_queue.Length)
    {
        q = my_queue.ElementAt(k++);

        foreach (int i in desc[q.v])
            if ( !mark[i] )
            {
                mark[i] = true;
                my_queue.Enqueue(new QueueElement()
                                 { v = i, d = q.d+1, p = k-1 });
                if (dict[i] == w2)
                {
                    DisplayPath();
                    return true;
                }
            }
    }
    return false;
}

public void CallWordPathFinder()
{
    dict = System.IO.File.ReadAllLines("dict.txt");

    int n = dict.Length;
    int i, j;

    w_res    = new List<string>();
    my_queue = new Queue<QueueElement>();
    mark     = new bool[n];
    desc     = new List<int>[n];

    for (i=0; i<n; i++)
    {
        desc[i] = new List<int>();
        mark[i] = false;
    }
```

```
            // ако две думи са съседни, ги правим съседни и в графа
            for (i = 0; i<n-1; i++)
                for (j=i+1; j<n; j++)
                    if (str_neigh(dict[i], dict[j]))
                    {
                        desc[i].Add(j);
                        desc[j].Add(i);
                    }
            if (!SearchWordsPath("math", "rose"))
                Console.WriteLine("Някоя от думите липсва в речника " +
                                  "или няма път между тях.");
            Console.ReadKey();
        }
    }
```

С търсенето в ширина намираме минималния път между двете зададени думи.

Търсене в дълбочина (Depth-first search)

Обхождането на графа става по следния начин: Започваме с връх от графа, маркираме го, че е посетен и обработваме информационната част на върха. След това намираме немаркиран връх наследник и извършваме предходната процедура с него. Ако нямаме немаркиран наследник, се връщаме към предходния връх и проверяваме за следващи наследници. Ако текущият връх няма повече непосетени наследници и няма предходник, трябва да проверим дали има други непосетени върхове. В този случай графът е несвързан.

Процесът приключва, когато намерим търсеното решение или нямаме повече върхове за обхождане.

Този алгоритъм е еднакъв с метода обхождане с връщане назад, но терминът *търсене в дълбочина* се използва, когато работим с графи.

За конкретната задача с търсенето в дълбочина ще намерим път между двете думи, ако има такъв, но този път може да не е минимален.

Ще използваме представянето на графа и някои от функциите от предходното решение.

```
string[] dict;
List<int>[] desc;
bool[] mark;

bool str_neigh(string a, string b);

int FindWord(string str);
```

За обхождането в дълбочина ще използваме рекурсивна функция. Към параметрите освен номера на върха е добавена и думата, до която търсим път. Функцията връща логическа стойност, която използваме за отпечатване на пътя при обратния ход на функцията.

```
bool DepthFirstSearch(int vert, string word)
{
    mark[vert] = true;
    if (dict[vert] == word) return true;

    bool found = false;
    int i = 0;

    foreach (int v in desc[vert])
        if (!mark[v])
            if (found = DepthFirstSearch(v, word))
            {
                i = v; break;
            }

    if (found)
        Console.WriteLine(dict[i]);

    return found;
}

bool SearchWordsPath(string w1, string w2)
{
    int k;

    k = FindWord(w1);
    if (k == -1) return false;

    k = FindWord(w2);
    if (k == -1) return false;

    if (!DepthFirstSearch(k, w1))
        Console.WriteLine("Няма път между думите.");
    else
        Console.WriteLine(w2);
    return true;
}
```

Функция CallWordPathFinder остава същата, като тази от търсенето в ширина.

Търсенето в дълбочина за тази задача не дава оптимален път от една до друга дума и в повечето случаи резултатът е много далече от оптималния. Но може да се използва, ако само ни интересува дали задачата има решение. В този случай ще отпадне извеждането на думите от рекурсивната функция.

6.4. Минимален път в граф

Алгоритъм на Дейкстра (Dijkstra)

Нека е дадена карта на пътно–транспортната мрежа на една държава. Населените места и свързващите ги пътища са описани в картата, като са дадени и дължините на пътищата, свързващи директно всеки два града. Целта на задачата е да бъде намерена дължината на най-кратките пътища от дадено населено място до всяко едно друго в държавата.

Приемаме, че пътно–транспортната карта е представена чрез граф, в който населените места са представени чрез върхове, а свързващите ги пътища чрез ребра. Всяко ребро има определено тегло – дължината на пътя. По този начин задачата може да бъде сведена до търсене на минимален път в граф.

Съществуват различни алгоритми за търсене на оптимални пътища в граф. Най–ефективният начин за намиране на всички оптимални пътища от фиксиран връх до всички останали в един граф е алгоритъма на Дейкстра.

Като помощни масиви дефинираме **adjacencyMatrix**, съдържащ матрицата на съседства на графа, **D**, съдържащ списъка с оптималните пътища, намерени до момента и **availableNodes**, съдържащ списък с необходените върхове. В целочислената променлива **NodesCount**, съхраняваме броя на възлите в графа.

- Решаваме задачата за възел ***i***;
- Инициализираме масива **D**[j], ***j = 1, … , NodesCount*** с наличните пътища от ***i*** до всички останали възли в графа. Ако от ***i*** до ***j*** няма път **D**[j] се инициализира с максималната стойност за типа на масива **D**;
- Инициализираме масива **availableNodes**, като маркираме всички възли за необходени;
- Маркираме връх ***i*** за обходен;
- Намираме връх ***j***, такъв че да бъде необходен и пътя от ***i*** до него да е най–краткия от намерените минимални пътища до момента;
- Ако няма такъв то задачата е решена и резултатът е в масива **D**;
- Маркираме върха ***j*** за обходен и за всички негови съседни необходени върхове проверяваме дали намереният до момента оптимален път до тях е по–дълъг от сумата на оптималния път до ***j*** и пътя от ***j*** до тях. Ако това е изпълнено значи сме намерили по–кратък път до съответния връх и стойността се записва в **D**;
- Повтаряме стъпки 5, 6, 7 и 8 не влезем в случая, описан в стъпка 6.

```
private int[,] adjacencyMatrix;
private int[] D;
private bool[] availableNodes;

public int NodesCount;

public void Solve(int node)
{
    for (int i = 0; i < NodesCount; i++)
    {
        if (adjacencyMatrix[node, i] <= 0)
            D[i] = int.MaxValue;
        else
            D[i] = adjacencyMatrix[node, i];

        availableNodes[i] = true;
    }
    availableNodes[node] = false;

    while (true)
    {
        int j = int.MaxValue;
        int di = int.MaxValue;

        for (int i = 0; i < NodesCount; i++)
        {
            if (availableNodes[i] && (D[i] < di))
            {
                di = D[i];
                j = i;
            }
        }

        if (j == int.MaxValue)
            break;

        availableNodes[j] = false;
        for (int i = 0; i < NodesCount; i++)
        {
            if (availableNodes[i] && (adjacencyMatrix[j, i] > 0))
            {
                if (D[i] > (D[j] + adjacencyMatrix[j, i]))
                {
                    D[i] = D[j] + adjacencyMatrix[j, i];
                }
            }
        }
    }
}
```

7. Лакоми алгоритми

Лакоми (алчни) алгоритми (*Greedy algorithms*) наричаме техника, която използваме за решаване на оптимизационни задачи. Изграждането на решението става последователно като на всяка стъпка правим най-добрия за момента избор. На по-късен етап може да са окаже, че този избор е неправилен. Лакомите алгоритми се съставят лесно, но не винаги гарантират оптималното или правилно решение. Независимо от това съществуват задачи, които се решават оптимално и други, за които решението не е оптимално, но се намира достатъчно бързо, приемливо е и може да се използва на практика.

Сума с минимален брой монети

Да се намери минималния брой монети (банкноти), с които може да се получи искана сума. Имаме неограничен брой монети, които са с предварително определена стойност.

Ще решим задачата по следния начин. Изваждаме от текущата сума стойността на монета с възможно най-голяма стойност по-малка или равна на сумата. Продължаваме по този начин докато изчерпим исканата сума. В конкретната реализация към текущата стойност на променливата за минималния брой монети ***min_coins*** добавяме резултата от делението на сумата с най-голямата текуща стойност на монета: ***min_coins += sum / k***, а новата стойност на сумата получаваме, като остатъка от разделянето на сумата със стойността на монетата ***sum = sum % k***.

Това решение може да не оптимално за определени стойности на монетите и за определени суми. Ако стойността на всяка следваща монета е поне два пъти по-голяма от предходната, решението ще бъде винаги оптимално. Например, за монети със стойности ***1, 2, 5, 10, 20, 50***.

Ако имаме монета със стойност по-малка от удвоената стойност на предходната монета, то оптималният резултат ще зависи от сумата, която искаме да получим. Например, имаме монети със стойности ***1, 2, 5, 7*** и сума ***10***. С лакомия алгоритъм ще получим резултат ***10 = 7 + 2 + 1***, т.е. използваме ***3*** монети. А можем да получим сумата с две монети: ***10 = 5 + 5***.

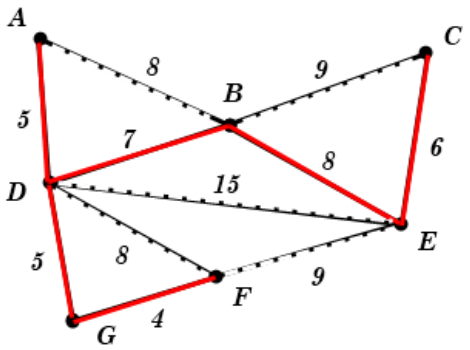
Независимо от този резултат, решението е кратко и бързо и може да се използва на практика.

```
void MinCoinsNumber()
{
    int sum, num_coins;

    Console.Write("Enter sum: ");
    sum = int.Parse(Console.ReadLine());
```

```
        Console.Write("Number of coin values: ");
        num_coins = int.Parse(Console.ReadLine());

        int[] coins_value = new int[num_coins];

        Console.WriteLine("Enter values of coins: ");
        for (int i = 0; i < num_coins; i++)
            coins_value[i] = int.Parse(Console.ReadLine());

        Array.Sort(coins_value);
        Array.Reverse(coins_value);

        int min_coins = 0;
        foreach (int k in coins_value)
        {
            min_coins += sum / k;
            sum = sum % k;
        }
        Console.Write("Min. number of coins: ");
        Console.WriteLine(min_coins);
    }
```

Минимално покриващо дърво

Нека са дадени няколко населени места в местност без пътна инфраструктура. Трябва да бъдат изградени пътища, свързващи населените места. За всяка двойка населени места се предоставя информация дали могат да бъдат свързани директно с път и ако да, каква ще бъде дължината му. Целта на задачата е да бъдат определени пътищата, които ще могат да свържат населените места, по такъв начин, че от всяко едно населено място да може да се стигне до всяко друго и същевременно дължината на изградените пътищата да бъде минимална.



Представяме населените места и възможните пътища с неориентиран граф с положителни тегла на ребрата. Търсим дърво, което съдържа всички върхове на изходния граф и сумата от ребрата на дървото е минимална (*минимално покриващо дърво*).

Решението на тази оптимизационна задача с лаком алгоритъм е винаги оптимално. Алгоритъмът още се нарича и алгоритъм на Прим (*Prim's algorithm*).

Първо намираме най-късото реброто (с най-малка стойност) и го добавяме към дървото. На всяка следваща стъпка от съседните ребра, непринадлежащи на дървото избираме това с минимална дължина. Добавяме това ребро към дървото и продължаваме, докато в дървото се включат всички върхове на графа.

В реализацията ще използваме матрица на съседства за представяне на графа. За реална задача, в която броя на ребрата не е много по-голям от броя на върховете и матрицата ще заеме много памет, ще трябва да използваме друго представяне. За представяне на дървото ще използваме множество, в което ще записваме върховете и списък, в който ще записваме ребрата на дървото. Програмата ще работи, ако графът е свързан.

```
public class MinimumSpanningTree
{
    int nodes_number;
    int[,] my_graph;

    class TEdge
    {
        public int node1 { get; set; }
        public int node2 { get; set; }
        public int value { get; set; }
    }
    HashSet<int> graph_nodes;
    HashSet<int> tree_nodes;
    List<TEdge> tree_edges;

    public MinimumSpanningTree()
    {
        Console.Write("Enter number of graph nodes: ");
        nodes_number = int.Parse(Console.ReadLine());

        my_graph = new int[nodes_number, nodes_number];
        graph_nodes = new HashSet<int>();
        tree_nodes = new HashSet<int>();
        tree_edges = new List<TEdge>();

        for (int i = 0; i < nodes_number; i++) graph_nodes.Add(i);

        for (int i = 0; i < nodes_number; i++)
            for (int j = 0; j < nodes_number; j++)
                my_graph[i,j] = int.MaxValue;

        for (int i = 0; i < nodes_number; i++)
            for (int j = i + 1; j < nodes_number; j++)
                my_graph[i,j] = my_graph[j,i] =
                    int.Parse(Console.ReadLine());
```

```
        int min = int.MaxValue;
        int n1 = -1, n2 = -1;

        for (int i = 0; i < nodes_number; i++)
            for (int j = i + 1; j < nodes_number; j++)
                if (my_graph[i,j] < min)
                {
                    min = my_graph[i,j];
                    n1 = i;
                    n2 = j;
                }

        tree_nodes.Add(n1);
        tree_nodes.Add(n2);

        tree_edges.Add(new TEdge()
            { node1 = n1, node2 = n2, value = my_graph[n1, n2] });

        graph_nodes.Remove(n1);
        graph_nodes.Remove(n2);

        while(tree_nodes.Count < nodes_number)
        {
            min = int.MaxValue;
            foreach (int i in tree_nodes)
                foreach (int j in graph_nodes)
                    if ( my_graph[i,j] < min )
                    {
                        min = my_graph[i, j];
                        n1 = i;
                        n2 = j;
                    }

            tree_nodes.Add(n2);

            tree_edges.Add(new TEdge()
                { node1 = n1, node2 = n2, value = my_graph[n1, n2] });

            graph_nodes.Remove(n2);
        }

        Console.WriteLine("Edges of minimal spanning tree: ");
        foreach(TEdge e in tree_edges)
        {
            Console.Write("node1: "); Console.WriteLine(e.node1);
            Console.Write("node2: "); Console.WriteLine(e.node2);
            Console.Write("value: "); Console.WriteLine(e.value);
        }
    }
}
```

Разходката на коня

Отново се връщаме към задачата от глава 4. При решението чрез обхождане с връщане назад, сложността на алгоритъма е прекалено висока и решението е неефективно. През 1823 Варнсдорф предлага евристичен начин за избор на следващия ход на фигурата. Правилото на Варнсдорф гласи, че конят винаги трябва да посещава свободна съседна позиция от най-ниска степен (позицията, от която имаме най-малко възможни ходове). Това решение е от типа на лакомите алгоритми. На всяка стъпка избираме най-добър за момента ход на коня (описания по-горе критерий). Останалите подслучаи на задачата биват игнорирани.

Използваме структурите, описани в първото решение на задачата, и

- Маркираме обхождaната позиция за посетена;
- Ако всички позиции на дъската са вече посетени, значи сме намерили решение и връщаме положителен резултат;
- Последователно обхождаме всички достижими позиции от мястото, където коня се намира в момента и намираме тази, от която коня има най-малко възможни ходове;
- Ако намерим такава позиция я обхождаме. Ако обхождането върне положителен резултат, значи е намерено решение и връщаме положителен резултат.

```
private int[] offsetX = new int[] { -2, -2,  1, -1,  2,  2,  1, -1 };
private int[] offsetY = new int[] {  1, -1,  2,  2,  1, -1, -2, -2 };
private int[ , ] visited;

public readonly int BoardSize = 8;

private bool AllNodesVisited()
{
    for (int i = 0; i < this.BoardSize; i++)
    {
        for (int j = 0; j < this.BoardSize; j++)
        {
            if (visited[i, j] <= 0)
                return false;
        }
    }
    return true;
}

private int PossibleMovesFrom(int x, int y)
{
    int result = 0;
    for (int i = 0; i < offsetX.Length; i++)
    {
        int newX = x + offsetX[i];
        int newY = y + offsetY[i];
```

```
        if (((newX > -1) && (newX < this.BoardSize)) &&
           ((newY > -1) && (newY < this.BoardSize)) &&
           (visited[newX, newY] <= 0))
        {
            result++;
        }
    }
    return result;
}

public bool Solve(int x, int y, int current)
{
    visited[x, y] = current;

    if ( AllNodesVisited() ) return true;

    int minPossibilities = int.MaxValue;
    int nextX = -1;
    int nextY = -1;

    for (int i = 0; i < offsetX.Length; i++)
    {
        int tempX = x + offsetX[i];
        int tempY = y + offsetY[i];

        if (((tempX > -1) && (tempX < this.BoardSize)) &&
           ((tempY > -1) && (tempY < this.BoardSize)) &&
           (visited[tempX, tempY] <= 0))
        {
            int possibilities = PossibleMovesFrom(tempX, tempY);

            if (minPossibilities > possibilities)
            {
                minPossibilities = possibilities;
                nextX = tempX;
                nextY = tempY;
            }
        }
    }

    if ((nextX > -1) && (nextY > -1) &&
        (Solve(nextX, nextY, current + 1)))
    {
            return true;
    }

    visited[x, y] = 0;
    return false;
}
```

8. Динамично оптимиране

Най-общо методът на динамичното оптимиране се свежда до следното: на всяка стъпка от решението на задачата намираме текущи оптимални стойности, като използваме намерените оптимални стойностите от предходната стъпка или всички намерени предходни оптимални стойности. Можем да използваме тази техника само ако можем да разделим задачата на по-малки подзадачи и успеем да формулираме, как ще получим следващите оптимални решения на подзадачите.

Сума с минимален брой монети

Да се намери минималния брой монети (банкноти), с които може да се получи искана сума. Имаме неограничен брой монети, които са с предварително определена стойност.

За разлика от решението с лаком алгоритъм от предходната глава тук ще намерим оптимално решение, независещо от входните данни.

Намираме последователно минималния брой монети за сумите от ***1*** нагоре. Нека ***MinCoins(k)*** е минималния брой монети за сума ***k***. Приемаме, че сме намерили всички стойности на ***MinCoins()*** до сумата ***k-1***. Намираме минималния брой монети за сума ***k*** по следния начин:

- Ако имаме монета със стойност ***k***, то ***MinCoins(k) = 1***;
- За всяка една монета със стойност ***p***, ***p < k***, можем да направим сумата ***k***, като ***k = (k-p) + p***. Намерили сме ***MinCoins(k-p)***, така че нашата сума я правим с ***MinCoins(k-p)+1*** на брой монети.
- Правим цикъл по стойностите на монетите с променлива ***p*** и намираме минималната стойност на ***MinCoins(k-p).*** Получаваме ***MinCoins(k) = min{ MinCoins(k-p) } + 1***, където ***p*** се променя по стойностите на монетите.

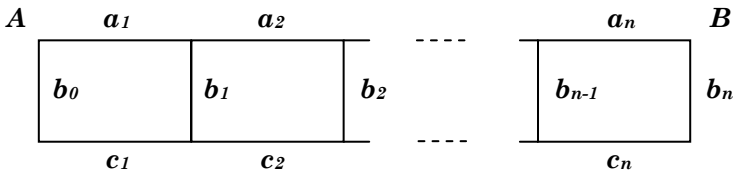
Т.е. намираме оптималната стойност на текущата стъпка с помощта на оптималните стойности от предходните стъпки.

```
static void Main(string[] args)
{
    int sum, num_coins;

    Console.Write("Enter sum: ");
    sum = int.Parse(Console.ReadLine());

    Console.Write("Number of coin values: ");
    num_coins = int.Parse(Console.ReadLine());
```

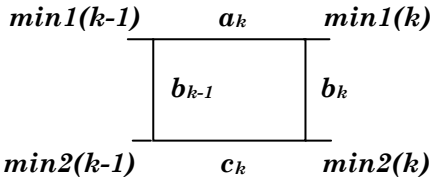
```
        int [] min_number = new int[sum+1];
        int [] coins_value = new int[num_coins];

        Console.WriteLine("Enter values of coins: ");
        for (int i = 0; i < num_coins; i++)
            coins_value[i] = int.Parse(Console.ReadLine());

        for (int k = 1; k <= sum; k++)
        {
            int min_sum_c, prev_min;
            min_sum_c = int.MaxValue;

            foreach (int p in coins_value)
            {
                if (k < p) continue;
                if (k == p)
                {
                    min_sum_c = 1; break;
                }
                prev_min = min_number[k - p] + 1;
                if (prev_min < min_sum_c) min_sum_c = prev_min;
            }
            min_number[k] = min_sum_c;
        }

        Console.Write("Min. number of coins: ");
        Console.WriteLine(min_number[sum]);
    }
```

Намиране на минимален път

Имаме следната квадратна мрежа, със зададени дължини на страните. Търсим минималния път от ***A*** до ***B***.



Търсим минималния път последователно като на всяка стъпка намираме минималния път до всеки две пресечни точки една над друга:



Ако сме намерили ***min1(k-1)*** и ***min2(k-1)***, намираме минималния път до следващите две точки като:

$$min1(k) = \min \{ min1(k-1) + a_k, min2(k-1) + c_k + b_k \},$$

$$min2(k) = \min \{ min1(k-1) + a_k + b_k, min2(k-1) + c_k \}.$$

В началото $min1(0) = 0$ и $min2(0) = b_0$. Минималният път до точка ***B*** е равен на ***min1(n)***.

```
static void Main(string[] args)
{
    int n;
    Console.Write("Enter n: ");
    n = int.Parse(Console.ReadLine());

    int[] a = new int[n+1];
    int[] b = new int[n+1];
    int[] c = new int[n+1];
    int[] min_path1 = new int[n+1];
    int[] min_path2 = new int[n+1];

    Console.WriteLine("Enter upper horizontal distances: ");
    for (int i = 1; i <= n; i++)
        a[i] = int.Parse(Console.ReadLine());

    Console.WriteLine("Enter vertical distances: ");
    for (int i = 0; i <= n; i++)
        b[i] = int.Parse(Console.ReadLine());

    Console.WriteLine("Enter down horizontal distances: ");
    for (int i = 1; i <= n; i++)
        c[i] = int.Parse(Console.ReadLine());

    min_path1[0] = 0;
    min_path2[0] = b[0];

    for (int k = 1; k <= n; k++)
    {
        min_path1[k] =
           Math.Min(min_path1[k-1] + a[k], min_path2[k] + c[k] + b[k]);
        min_path2[k] =
           Math.Min(min_path1[k-1] + a[k] + b[k], min_path2[k] + c[k]);
    }

    Console.Write("The minimal path from A to B: ");
    Console.WriteLine(min_path1[n]);
}
```

Оптимизиране на приходи

Малка компания произвежда мебели по предварителна заявка. За дадено време се изпълнява само една заявка, без прекъсване на работата по нея. Всяка заявка се изпълнява за цяло число дни и има предварително зададена цена (цяло число). Компанията разполага с ***n*** на брой свободни работни дни и има списък с ***k*** на брой предварителни заявки. Намерете максималния приход, който може да се получи за тези ***n*** работни дни, като изпълнените заявки трябва да са цяло число.

По същия начин, както в предходните две задачи, ако знаем максималните приходи за ***1, 2, ... , n-1*** на брой дни, трябва да намерим оптималния приход за ***n*** на брой работни дни. Задачата прилича на тази с монетите, но тук търсим максимум и ако за поредния номер ***k*** на дните има заявка за толкова дни, няма да я взимаме за директна стойност. Другата съществена разлика е, че можем да взимаме всяка заявка само по един път. Затова декларираме масив **used_requests** от тип **HashSet<int>**, в който записваме номерата на използваните заявки.

```
public class TRequest
{
    public int necessary_days { get; set; }
    public int request_value { get; set; }
    public int number { get; set; }

    public TRequest (int n)
    {
        this.number = n;
    }
}

static void Main(string[] args)
{
    int days_num, requests_num;

    Console.Write("Enter number of days: ");
    days_num = int.Parse(Console.ReadLine());

    Console.Write("Number of requests: ");
    requests_num = int.Parse(Console.ReadLine());

    int[] max_value = new int[days_num + 1];
    HashSet<int>[] used_requests = new HashSet<int>[days_num + 1];

    for (int i = 0; i <= days_num; i++)
        used_requests[i] = new HashSet<int>();
```

```
    TRequest[] request = new TRequest[requests_num];

    for (int i = 0; i < requests_num; i++)
        request[i] = new TRequest(i);

    Console.WriteLine("Enter necessary days and value of request: ");
    for (int i = 0; i < requests_num; i++)
    {
        request[i].necessary_days = int.Parse(Console.ReadLine());
        request[i].request_value  = int.Parse(Console.ReadLine());
    }

    for (int k = 1; k <= days_num; k++)
    {
        int max_sum = 0, cval, req_num = 0, prev_day;

        foreach (TRequest req in request)
        {
            if (k < req.necessary_days) continue;

            prev_day = k - req.necessary_days;
            if (used_requests[prev_day].Contains(req.number)) continue;

            cval = max_value[prev_day] + req.request_value;
            if (cval > max_sum)
            {
                max_sum = cval;
                req_num = req.number;
            }
        }
        max_value[k] = max_sum;
        used_requests[k].Add(req_num);
    }
    Console.Write("Max. possible value: ");
    Console.WriteLine(max_value[days_num]);
}
```

Литература

1. Амерал Л., *Алгоритми и структури от данни в C++*, Софтех, 2001.
2. Азълов П., *Обектно-ориентирано програмиране – структури от данни и STL*, Сиела, София, 2008.
3. Крушков Хр., *Програмиране на C++, I част – Въведение в програмирането*, 3-то преработено и допълнено издание, Коала прес, 2010.
4. Наков П., П. Добриков, *Програмиране = ++Алгоритми;*, трето издание, TopTeam Co., София, 2005.
5. Наков С., В. Колев и колектив, *Въведение в програмирането със C#,* Фабер, Велико Търново, 2011.
6. Рахнев А., К. Гъров, О. Гаврилов, *Бейсик в задачи,* АСИО, София, 1995.
7. Шишков Д., М. Върбанов, А. Голев и колектив, *Структури от данни*, Интеграл, Добрич, 1995.
8. Bentley J., *Programming pearls,* Addison-Wesley, 1986. (руски превод: Бентли Д., Жемчужиньь творчества программистов, Радио и связ, Москва, 1990)
9. Brassard G., P. Bratley. *Algorithmics, Theory and Practice*, Prentice-Hall, New Jersey, 1988.
10. Cormen T., C. Leiserson, R. Rivest, C. Stein, *Introduction to algorithms*, Third edition, MIT Press, 2009.
11. Sedgewick R., K. Wayne, *Algorithms*, 4th edition, Addison-Wesley, 2011.
12. Schildt H., *The Complete reference C# 4.0*, McGraw-Hill, 2002.
13. Troelsen A., *Pro C# 2010 and the .NET 4 Platform*, F'ifth Edition, Apress, 2010.
14. http://msdn.microsoft.com/en-us/library

УЧЕБНО ПОМАГАЛО ПО АЛГОРИТМИ И ПРОГРАМИ СЪС C#

Ангел Голев

Българска, първо издание

Печат и подвързия: УИ „Паисий Хилендарски”

Пловдив, 2012

ISBN 978-954-423-791-2