"Лакоми" алгоритми
(*"алчни"* или *greedy* алгоритми)

При решаването на различни оптимизационни задачи досега сме търсили точни или оптимални решения. В повечето случаи се търсенето ставаше измежду всички възможни решения, което изисква много пресмятания, или (в случая, ако използваме динамично оптимиране) – допълнителна памет.

Съществува, обаче клас алгоритми, които не винаги дават точното решение, но за сметка на това са много прости и лесно реализуеми (т. н. *евристични алгоритми*). Такъв алгоритъм обикновено използва следната идея: насочва се към един от всичките подслучаи на задачата и решава единствено него с надеждата, че той ще се окаже правилният. Изборът на този случай се извършва въз основа на локален критерий за оптималност.

"Лакомите " алгоритми са подклас от евристичните и, както подсказва и названието им, се насочват винаги към най-добрия за момента избор, погледнато локално, като на по-късен етап може да се окаже, погледнато глобално, че този избор не е бил най-подходящ.

(Евристични алгоритми - припомняне)

Свойства:

- дават добри, но не винаги оптимални решения;
- могат да бъдат реализирани бързо и лесно

Нямат универсален подход за създаването им (“Еврика!”).

Препоръки:

Всички изисквания към решението се разделят на 2 групи:

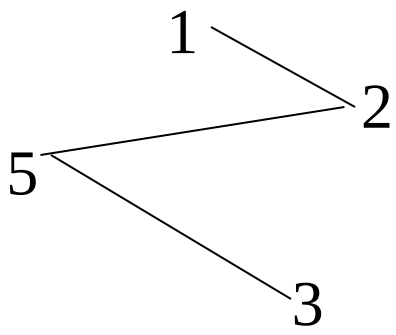
1. Тези, които лесно се постигат
2. Тези, които не се постигат лесно.

или

1. Тези, които са задължителни
2. Тези, за които може да се направи компромис.

Целта е да се изгради алгоритъм, гарантиращ изпълнение на изискванията от 1 група и не гарантиращ тези от 2-та (което не означава, че няма да се правят опити да се изпълнят и те).

Задача за търговския пътник:

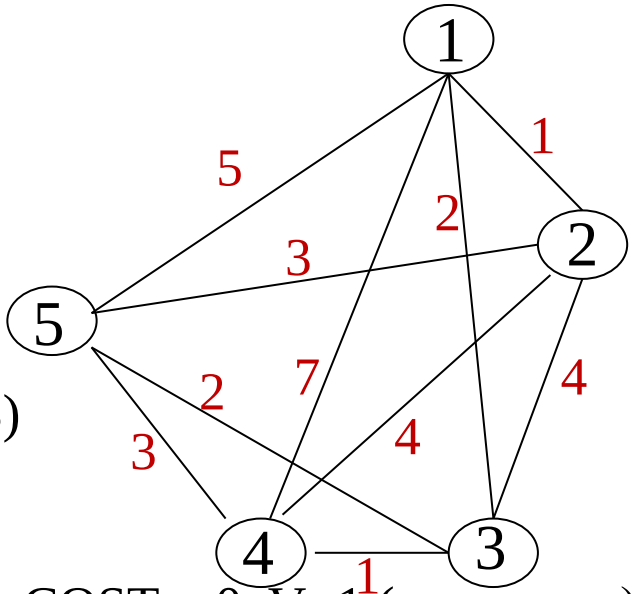


2 – 5 (3)

5 – 3 (2)

3 – 4 (1)

4 – 1 (7) = 14 (MIN = 13)



Стъпка:

0 (инициализация) LIST = 0; COST = 0, V=1 (нач. възел)

1 (посещение на всички градове) За **i** от 1 до **n**-1
повтаряй стъпка 2;

2 (избор на следващо ребро) (V,W) е ребро с MIN цена, свързващо V с "непосетен" възел W:
LIST=LIST+W;
COST=COST+стойност(V,W);
отбелязване на W като посетен;
V=W;

3 (завършване на маршрута) LIST=LIST+1;
(добавяне на нач. град)
COST=COST+стойност(V,1);

4 Край

Сложност на алгоритъма е $O(n^2)$!

"Лакомите" алгоритми се съставят лесно, реализацията им не е сложна (като на всеки евристичен алгоритъм). Единственият им недостатък, че не винаги гарантират правилното решение на задачата, макар че намереното винаги е близко до оптималното.

Пример за "лаком" алгоритъм: Да се намери начин за получаване на дадена сума m (m – цяло число), като се използва минимален брой банкноти с номинали от множеството {1, 2, 5, 10, 20, 50, 100} лева за българската национална валута. Един от възможните подходи при търсене на решението е:

1. Инициализираме текуща сума $s = 0$;
2. Вземаме банкнота i с max стойност a_i от множеството, такава че $s+a_i <= m$
   - 2.1. Ако няма банкнота, за която $s+a_i <= m$, задачата няма решение. Край.
   - 2.2. Иначе, вземаме банкнота i и увеличаваме s с a_i
     - 2.2.1. Ако $s = m$ задачата е решена. Край.
     - 2.2.2. Ако $s < m$ повтаря се стъпка 2

Например, сумата 197 лева ще се плати последователно с 1 банкнота от 100 лв., 1 от 50 лв., 2 по 20 лв., 1 от 5 лв. и 1 от 2 лв.

Описаният алгоритъм отговаря на критериите на "лаком" алгоритъм: на всяка стъпка той избира максималната по стойност банкнота, като по този начин се стреми да постигне най-бързо търсената сума (което е така за горепосочения пример).

Но работи ли алгоритъма по същия начин при произволни входни данни?

Нека ***m*** =40 лв., а множеството банкноти да е {2, 5, 20, 30} лева. "Лакомият" алгоритъм ще даде следното решение: 30 + 2*5, т. е. сумата ще се плати с 3 банкноти. Очевидно, съществува и по-добро решение: 2*20. При същото множество номинали, но при друга сума (например, ***m*** =6 лв.), алгоритмът изобщо няма да намери решение, макар че то съществува: 5+2 > ***m*** , но ***m*** =2+2+2=6 и може да се плати с 3 банкноти.

Все пак подходът е ефективен и съществуват редица класически задачи, за които "лакомият" алгоритъм винаги намира вярно решение.

Задача за Египетските дроби

Древните египтяни са използвали само дроби с числител 1. Всяка друга дроб се е представяла и записвала като сума от дроби с числител 1.

Например, 7/9:

7/9 = 1/3 + 1/3 + 1/9

7/9 = 1/2 + 1/4 + 1/36

7/9 = 1/9 + 1/9 + 1/9 + 1/9 + 1/9 + 1/9 + 1/9

Условие на задачата: дадени са две естествени числа p и q ($q \neq 0$, $p < q$; $p, q \in N$). Да се намери представяне на дробта p/q във вида на сума:

$$p/q = 1/a_1 + 1/a_2 + \ldots + 1/a_n$$

при което знаменателите да бъдат различни:

$$a_i \neq a_j, \quad 1 \leq i, j \leq n, \quad i \neq j, \quad a_i \geq 2, \quad a_j \geq 2, \quad a_i, a_j \in N$$

Възможно е задачата да има повече от едно решение. Например:

$3/7 = 1/3 + 1/11 + 1/231$

$3/7 = 1/4 + 1/8 + 1/19 + 1/1064$

В нашия конкретен случай ще търсим произволно решение, отговарящо на условието знаменателите на намерените дроби да бъдат различни.

"Лаком" алгоритъм за решаване на задачата: на всяка стъпка поредния член в сумата да бъде максималната дроб, която може да се добави към текущата сума така, че резултатът да не надвишава p/q. Например, за p/q = 7/9 най-голямата дроб е 1/2. След това избираме дроб $1/a_2$, така че:

$1/2 + 1/a_2 \leq 7/9$, т. е. $1/a_2 \leq 7/9 - 1/2$ или $1/a_2 \leq 5/18$

Най-голямата дроб, отговаряща на това условие, е 1/4, тогава:

$$1/a_3 \leq 7/9 - 1/2 - 1/4 = 5/18 - 1/4 = 2/72.$$

Т. е. максималното a_3 е 1/36, с което сумирането приключва (1/2 + 1/4 + 1/36 =7/9).

Описание на алгоритъма:

while (p>1)

{

*Намира се максималната дроб **1/r**, ненадвишаваща **p/q**;*

*Отпечатва се дробта **1/r**;*

***p/q** = **p/q** - **1/r**;*

}

Необходимо е да уточним два момента:

- как да търсим се максималната дроб ***1/r***,

ненадвишаваща ***p/q*** (***q*** ≠ 0), т.е. минималното ***r*** , за което е изпълнено условието

$$1/r \le p/q\ , \quad r \ge 2,\ q \ge 2,$$

Това е еквивалентно на $r \ge p/q$. За да намерим ***r*** , можем да извършим целочислено деление ***p/q*** и да вземем най-малкото цяло число, не по-малко от ***p/q*** :

$$r = (q + p)\ /\ p$$

- разликата ***p/q*** = ***p/q*** - ***1/r*** се пресмята чрез привеждане под общ знаменател. Така, новите стойности ще бъдат:

```
p = p*r - q;
q = q*r;
```

Възможно е, след пресмятане на разликата да се получи съкратима дроб. Това ще попречи на правилната работа на алгоритъма единствено в случая, когато се получи дроб ***p/q*** , която може да се съкрати до ***1/x*** , т. е. ***q % p*** = = 0. В този случай, ако не се извърши съкращението, условието ***p*** > 1 ще продължава да бъде изпълнено и търсенето на дроби ще продължи до безкрайност.

В приведената по-долу реализацията съкращението извършва функцията *cancel():*

```
#include <iostream>

void cancel (unsigned long *p, unsigned long *q)
      // ако q е кратно на p, се извършва съответното
съкращение
{
          if (0 ==*q %* p)
          {
              *q /=*p;
              *p=1;
          }
}
```

```
void solve (unsigned  long p,    unsigned  long q)
{
        cout<<p<<"/"<<q<<'=';
        cancel(&p, &q);
        while (p>1)
        {   unsigned  long r;   //търсене на MAX дроб  1/r,  1/r<=p/q
            r = (p + q) / p;
            cout<<"1/"<<r<<'+';
            //изчисляване на    p/q - 1/r
            p=p*r - q;
            q=q*r;
            cancel(&p, &q);
        }
        if (p >0)
                cout<<p<<'/'<<q;

}
void main()
{
        solve(3,7);
}
```

Да разгледаме една такава задача, свързана с максимално съчетание на дейности:

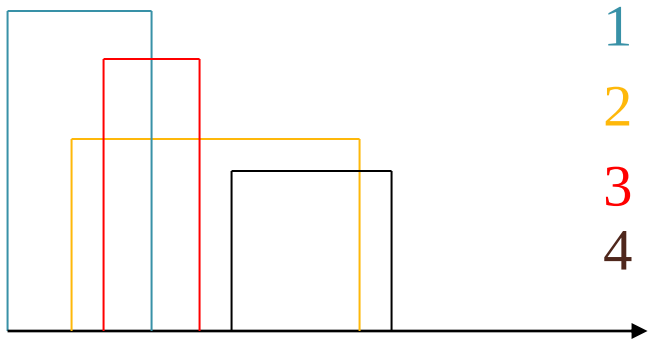
Задача: Има n дисциплини, които се четат през даден семестър и искат да ползват една и съща зала (за по-лесно ще казваме - n лекции). Всяка лекция i се характеризира с две цели числа – началния час s_i и крайния час f_i. Нека те да са естествени числа и всяка лекция може да продължава повече от 2 часа. Необходимо е максимално да натоварим залата, т. е. трябва да се изберат максимален брой лекции, така че никои две от избраните да не се провеждат в едно и също време (в даден момент t в залата може да се чете най-много една лекция i при условия $s_i <= t <= f_i$ и $1 <= i <= n$).

Задачата е класически пример за ефективно и правилно работещ "лаком" алгоритъм .

Нека да имаме следната конфигурация от 4 лекции:

лекциите 1 и 4 не се пресичат във времето

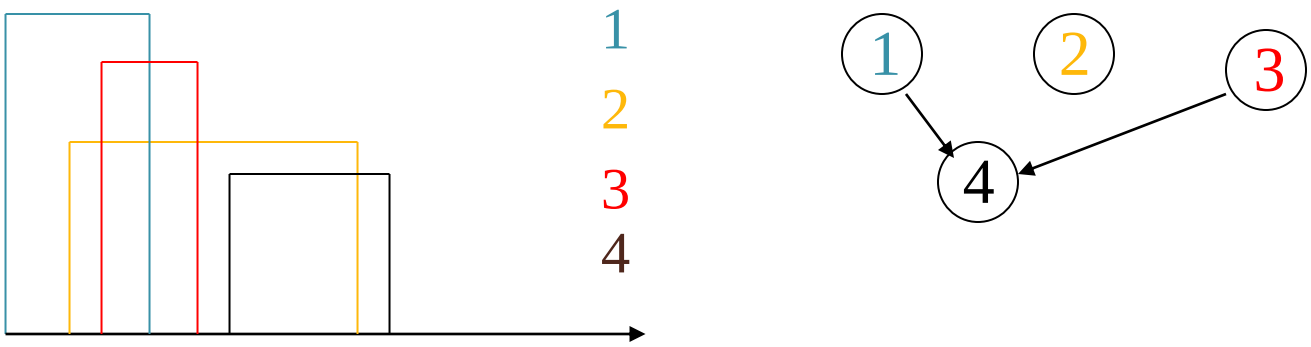
1 и 2, 2 и 3, 3 и 4 не могат да се провеждат едновременно



Възможни са следните подходи:

Алгоритъм 1:

Построяваме ориентиран граф $G(V,E)$ с n върха, в който на всеки две “непресичащи” се лекции i и j (т. е. $f_i<s_j$) съответства реброто (i, j) от следния граф:



Ако намерим най-дълъг път в G (алгоритмът вече е разгледан - търсене на най-дълъг път в ацикличен граф), ще получим решение със сложност $O(n^2)$.

Алгоритъм 2:

Възможно е задачата да се реши без да се изгражда граф, като се използва динамично оптимиране. Дефинираме целева функция ***F***, оптимизираща броя на избраните лекции за време от 7 до ***t*** ч. :

- $F(t)=0$, ако не съществува лекция i, за която $f_i < t$
- $F(t)=\max\{F(s_i)+1\}$ в противен случай

$i=1, 2, \ldots, n$

$f_i < t$

Реализацията на алгоритъма ще изисква допълнителна памет за запомняне на междинни резултати при пресмятане на рекурентната формула. Сложността на алгоритъма е квадратична, размерът на допълнителната памет зависи линейно от ***n***.

Алгоритъм 3:

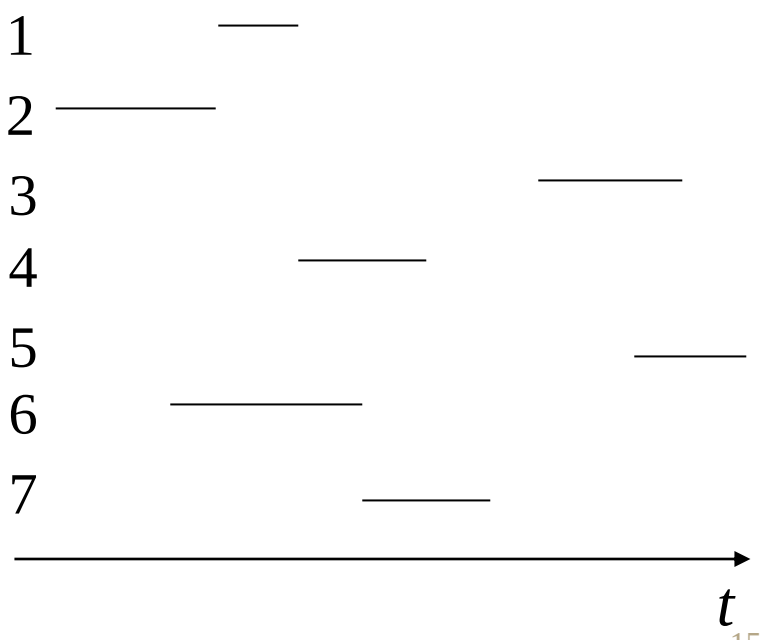
Задачата може да се реши просто и ефективно с "лаком" алгоритъм, при което се постига сложност $O(n*log_2 n)$, без да се използва допълнителна памет. Сложността на алгоритъма дори е линейна, но лекциите трябва да се сортират във възходящ ред по f_i, така че крайната сложност като цяло се определя от алгоритъма за сортиране.

Алгоритъм:

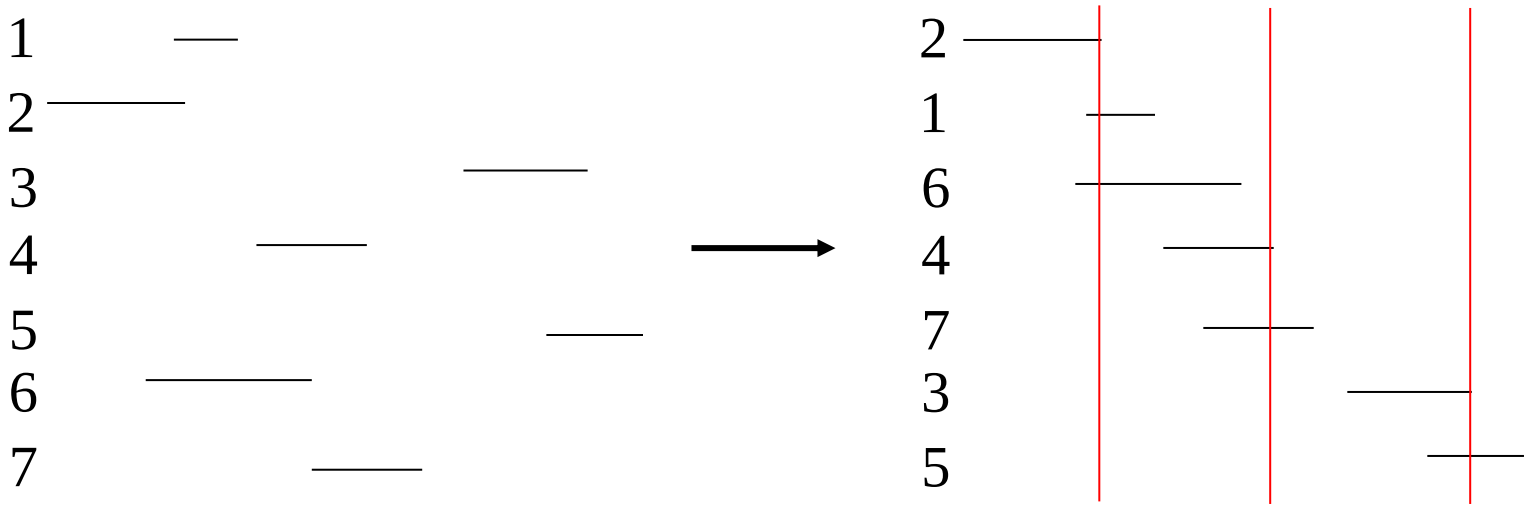
Разглеждаме лекциите от първата към последната като ги представяме разпределени във времето така (n=7):

1. Избира се лекция i, за която f_i е минимално (в началото винаги се избира първата, т. к. лекциите се сортират в нарастващ ред по f_i).
2. Всички лекции j, за които $s_j <= f_i$ (т.е. които се засичат с избраната) се изключват от разглеждането.

Повтаря се стъпка 1, докато се разгледат всички лекции.



Лекциите се сортират по f_i :



"Лакомият" алгоритъм ще избере лекция 2, след което ще пропусне лекции 1 и 6, ще избере 4, ще прескочи 7, ще избере 3 и ще пропусне 5. Полуформалното описание на алгоритъма е следното:

```
Избира се лекция 1;
i=1; j=1;
while (j<=n)
  {j++; if(s[j] > f[i])
          {избира се лекция j; i = j;}
  }
```

Пълната реализация на алгоритъма може да се види в Преслав Наков, Панайот Добриков. *Програмиране = ++Алгоритми*, TopTeam Co., София, 2002.

Използването на разгледания подход дава много добри решения на такива класически задачи като:

- Минимално оцветяване на граф и дърво
- Намиране на минимално покриващо дърво за даден граф ***G***
- Дробна задача за раницата
- Съставяне на процесорно разписание
- Кодиране на данни (код на Грей)

и много други.

Вероятностни алгоритми

За разлика от детерминираните алгоритми, които дават едни и същи резултати при даден комплект входни данни, *вероятностните* алгоритми могат да имат различни варианти решения. Ако в един алгоритъм има случаен избор (обикновено това са различни генератори на случайни числа), тогава той се нарича *вероятностен*.

Съществуват различни видове вероятностни алгоритми. Най-често срещаните са генератори на случайни числа, представляващи алгоритми за формиране на поредици от случайни числа в зададен интервал при определен закон за разпределение. Всеки език за програмиране обикновено съдържа в своите стандартни библиотеки подобни програми-генератори (например, в C/C++ - rand(), random() в stdlib.h).

Нека да разгледаме една примерна реализация на такъв генератор.

Най-често се използват генератори на равномерно разпределени случайни числа в интервал **[0, 1]** или в зададен целочислен интервал **[0, M-1].**

Една често използвана формула е

$$y_i = (a*y_{i-1}+c) \% m$$

където **a** и **c** са подходящо подбрани константи, y_o е началната стойност, която се задава от програмиста/потребителя, а получаваните стойности y_1 y_2 и т. н. са поредните случайни числа в интервала **[0, m].**

Реализация на генератор на равномерно разпределени случайни числа:

```
#include <iostream>
using namespace std;
#include <conio.h>
char R;          //работна променлива
int m,           //горната граница на интервала
    n,           //броят на числата
    a, c;        //константи
void main ()
{ int  y=1;      //начална стойност на y
  cout<<"\nЗадайте дължина на поредицата: ";
  cin>>n;
  cout<<"\nЗадайте m:";
  cin>>m;
  cout<<"\nКонстанта a:"; cin>>a;
  cout<<"\nКонстанта c:"; cin>>c;
  cout<<"\ny=1. Искате ли друга стойност на y (Y/N)?";
  cin>>R;
  if (R=='y'||R=='Y')
     { cout<<"\nY="; cin>>y;}
  for (int i=0; i<n; i++)
    { y=(a*y+c)%m; cout<<"\nrandom["<<i<<"]="<<y;}
}
```

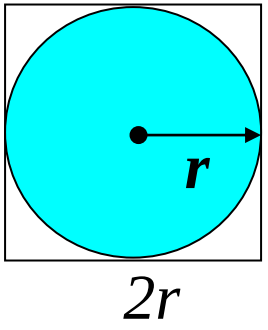
Получените числа са псевдослучайни.

Видове вероятностни алгоритми:

- Числени вероятностни алгоритми
- Алгоритми Монте Карло
- Алгоритми Лас Вегас
- Алгоритми Шеруд

1. Числен вероятностен алгоритъм за определяне на π¶

Лицето S на окръжност с радиус r може да се намери по формулата $S = \pi * r^2$. Отношението на лицето на окръжността към лицето на квадрата със страна $2r$ (в който може да се впише окръжността) е $\pi * r^2 / (2r)^2$. Вероятността p произволна точка, намираща се в квадрата, да бъде същевременно в окръжността, също е равна на $\pi * r^2 / (2r)^2$.



Да пресметнем тази вероятност емпирично: ще изберем ***t*** произволни точки в квадрата и ще проверим, колко от тях се намират и в окръжността (една точка е в окръжността, ако разстоянието от нея до центъра е по-малко от радиуса на окръжността). Полученият брой ***k***, разделен на ***t***, дава приближение на вероятността ***p***. Оттук следва:

$$\frac{k}{t} \approx \frac{\pi * r^2}{(2r)^2} \quad \text{или} \quad \pi \approx \frac{4 * k * r^2}{t * r^2} = \frac{4k}{t}$$

Вижда се, че приближената стойност на π ще зависи от отношението на броя на точките, попаднали в окръжността, към броя на проведените тестове ***t***. Точността ще бъде толкова по-голяма, колкото е по-голямо ***t*** и по-малко ***r*** (при по-малко ***r*** точността при деление ще бъде по-голяма, а при голям брой тестове ***t*** ще разполагаме с по-голяма статистическа извадка). Поради това, за радиуса ***r*** ще изберем голямо число, така че r^2 да се побира в типа *long*:

```
//Числен вероятностен алгоритъм за изчисляване на числото PI:
#include <iostream>
using namespace std;
#include <stdlib.h>
#include <math.h>                    // за sqrt( )
void main()
{
        long t=1000000;      // брой тестове
        long r=200;          //радиус на окръжността
        long r2=r/2;
        long k=0, i;

        for (i=0; i<t; i++)
        {
                long a=(rand() % r) - r2 + 1;
                long b=(rand() % r) - r2 + 1;
                if (sqrt (a*a + b*b) <= r2)  k++;
        }
        cout<<" PI = "<< (4.0*k)/t;
}
```

За t = 10 000 PI = 3.146, за t = 100 000 PI = 3.14268,
за t = 1 o00 000 PI = 3.1447

2. Алгоритми Монте Карло

Алгоритмите Монте Карло винаги "претендират", че са намерили решение на задачата. Понякога обаче, това решение се оказва невярно (в частност, ако разглеждаме оптимизационна задача, възможно е намереното от алгоритъма решение да бъде близко до оптималното, но все пак да съществува по-добро от него).

По-подробното описание на подхода може да се намери в Преслав Наков, Панайот Добриков. *Програмиране = ++Алгоритми*, TopTeam Co., София, 2002 и в Стойчо Стойчев, *Синтез и анализ на алгоритми*, "БПС", София, 2003.

Нека да разгледаме един класически пример за приложение на алгоритъма Монте Карло - проверка, дали дадено число ***n*** е просто. Ако изберем ***k*** естествени числа $p_1, p_2, \ldots, p_k$ $(2 <= p_i <= \sqrt{n})$ и ***n*** се дели на някое от тях, следва, че ***n*** не е просто (съставно). Така получихме един съвсем прост алгоритъм Монте Карло:

Алгоритъм Монте Карло 1:

```
for (i=1, …, k)
    if (n % (random(sqrt(n))+2)==0)
      {return n_е_съставно;}
     return n_е_просто;
```

Ако ***n*** във фрагмента е съставно, резултатът със сигурност ще е правилен. Ако ***n*** е просто, резултатът ще е верен с определена вероятност, която може да бъде пресметната [Преслав Наков, Панайот Добриков. *Програмиране = ++Алгоритми*, TopTeam Co., София, 2002].

Описаният алгоритъм е първата проста илюстрация на метода Монте Карло. По-долу ще бъдат разгледани още два (значително по-ефективни) алгоритъма за решаване на същата задача.

Първият от тях е следствие на малката теорема на Ферма:

Теорема на Ферма.

Нека n е просто число. Тогава $a^{n-1} \% n = 1$ за всяко $a = 1,2,\ldots, n-1$.

Теоремата дава следния практически резултат: Ако за дадено естествено число n намерено естествено число a $(1<a<n)$, такова че $a^{n-1} \% n <> 1$, то следва, че n е съставно.

Алгоритъм Монте Карло 2:

```
for (i=1, 2, … k)      //k опита
   {
     a=random(n-1) + 1;
          if (a^(n-1) % n != 1)
            {return n_е_съставно;}
   }
           return n_е_просто;
```

Тук отново не можем директно да определим вероятност ***p***, за която Алгоритъм 2 дава правилно решение.

Следва третия алгоритъм, чиято правилност може да бъде определена. Той е един от примерите за това, колко полезни и ефективни могат да бъдат алгоритмите Монте Карло.

Нека е дадено естествено число $n>4$. Да го представим във вида $2^s * t - 1$, където s и t са естествени числа, $s>0$, t е нечетно. Дадено е още едно естествено число a, $1<a<n-1$. Числото n се нарича *строго случайно при основа* a, ако $a^t \% n = 1$ или съществува естествено число i ($0<=i<s$) такова, че $a^{2i} \% n = n-1$

Теорема. Ако при произволна фиксирана база a ($1<a<n-1$) n не е строго случайно число, то n е съставно. В противен случай n е просто с вероятност, по-висока от 0.75.

Алгоритъм Монте Карло 3:

```
for (i=1, 2, … k) //k опита
   {
     a=random(n-3) + 2;
          if (n не е строго случайно при база a)
            {return n_е_съставно;}
   }
           return n_е_просто;
```

Теоремата твърди, че вероятността за намиране на правилното решение е поне 0,75. Така, ако изпълним Алгоритъм 3 за ***k***=4 (т.е. повторим основния опит 4 пъти), ще получим резултат за **n** с вероятност над 99%. Това превръща алгоритъма в изключително мощно средство за проверка и търсене на прости числа - с него може да се провери, дали едно число, с повече от 1000 цифри е просто, като вероятността за грешка е по-ниска от 10^{-100} .

За реализацията на Алгоритъм 3 трябва да се извърши проверка, дали ***n*** е строго случайно при произволно избрана база ***a***. Това става по следния начин:

- Намираме числата ***s*** и ***t***:

```
s=1; t=n-1;
while (t % 2 !=1)  {s++; t/=2;}
```

- Проверяваме, дали a^t % n = 1:

```
x=power(a, t) % n;
if (x==1) return (n_е_строго_случайно);
```

– Проверяваме, дали съществува i ($0<=i<s$), за което a^{2i} % n = n-1:

```
for (i=1, 2, … k)
    {
      if (n-1==x)
      return(n_е_строго_случайно);
      x=x*x%n;
    }
return (n_е_съставно);
```

Следва програма, която определя по метода Монте Карло (Алгоритъм 3), дали дадено цяло число ***n*** е просто или съставно:

```
#include <stdlib.h>  //rand()
#include <iostream>
#include <conio.h>

const unsigned k=10;
unsigned long bigmod (unsigned long a,
                      unsigned long t ,unsigned long n)

{return (t==1)?(a%n):(bigmod(a,t-1,n)* (a%n))%n;}
```

```
char strongRandom(unsigned long n, unsigned long a)
{
   unsigned long s=1, t=n-1, x, i;
   if(n<2) return 0;     //частен случай
   if (n==2) return 1;

   while (t%2!=1)        //стъпка 1
     { s++; t/=2;}
   x=bigmod(a,t,n);      //стъпка 2
   if (x==1) return 1;
   for (i=0; i<s; i++)   //стъпка 3
    {
     if (x==n-1) return 1;
     x=x*x%n;
    }
   return 0;
}
```

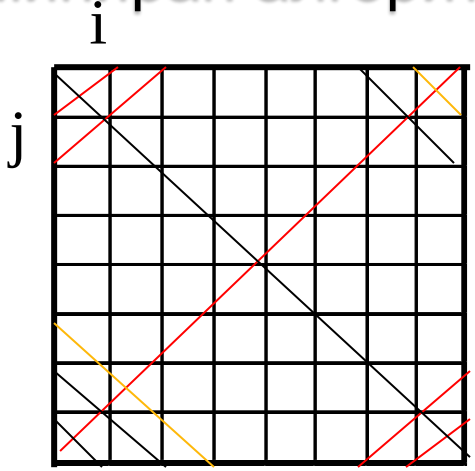
```
char isPrime (unsigned long n)
{
  unsigned long i;
  for (i=1; i<k; i++)
    {
      int a = rand() % (n-3) + 2;
      if (!strongRandom(n, a)) return 0;
    }
  return 1;
}
void main()
{
  unsigned long num;
  char c;
  do
  {
    cout<<"\nInput n:";
    cin>>num;
    cout<<"\nЧислото"<<num;
    if(isPrime(num))  cout<<" е просто";
      else cout<<" е съставно";
    cout<<"\nДруго n (Y/N):";
    cin>>c;
  }
  while (c=='Y'||c=='y');
}
```

3. Алгоритми Лас Вегас

Алгоритмите Лас Вегас не винаги намират решение, но когато намерят такова, то със сигурност е вярно (съответно ще е оптималното, ако разглеждаме оптимизационна задача).

Пример: Задачата за 8-те царици

А) Детерминиран алгоритъм с връщане



Б) Вероятностен алгоритъм Лас Вегас: знаейки, кои полета не са заети в даден момент, с генератор на случайни числа се избира едно от тях, след което този избор продължава, докато се намери решение или пък докато се достигне до ситуация, когато решение още не е намерено, но разполагането на следващата царица не е възможно. Тогава действията се повтарят отначало.

Алгоритми на Шеруд (Sherwood algorithms)

- Винаги намират верен резултат и са надеждни колкото детерминистичните алгоритми. Решават задачата, но не гарантират добра времева сложност.
- Обикновено се използват за ускоряване, когато детерминистичните алгоритми дават лоши резултати за асимптотичната оценка O(f(n)).

Вероятностен алгоритъм за Бързо сортиране (Randomized QuickSort)

Задача:

Вход:

Последователност от n числа $\langle a_1, a_2, \ldots, a_n \rangle$

Изход:

Пренареждане на числата $\langle a'_1, a'_2, \ldots, a'_n \rangle$ от входната последователност така, че

$$a'_1 \leq a'_2 \leq \ldots \leq a'_n$$

Идея на QuickSort:

```
QuickSort(a, p, r)
1    if p < r then
2            q← Partition(a, p, r)
3            QuickSort(a, p, q - 1)
4            QuickSort(a, q + 1, r)

QuickSort(a, 1, length[a])
```

Функция "Деление"

```
Partition(a, p, r)
1   x ← a[r]
2   i ← p - 1
3   for j ← p to r – 1 do
4       if a[j] ≤ x then
5           i ← i + 1
6           swap a[i] ↔ a[j]
7   swap a[i + 1] ↔ a[r]
8   return i + 1
```

Сложност на QuickSort

- Най-лошия случай на разделянето на масива (partitioning)

  $= O(n^2)$

- Най-добрия случай на разделянето на масива (partitioning)

  $= O(n \log n)$

Сложност на QuickSort

- Сложността на QuickSort е $O(n^2)$ в най-лошия случай, ако данните във входния масив са вече подредени:

  - Рандомизирането може да помогне да се намали вероятността на такава ситуация.
- Рандомизирането на QuickSort може да се реализира по два начина:

  - Случайно разбъркване на входния масив в началото.

  - Случаен избор на главния елемент (pivot value) на всяка стъпка на сортирането.
- Забележете разликата:

  едно е рандомизиране на данни, друго е да се рандомизира алгоритъм.

Рандомизирана версия на QuickSort

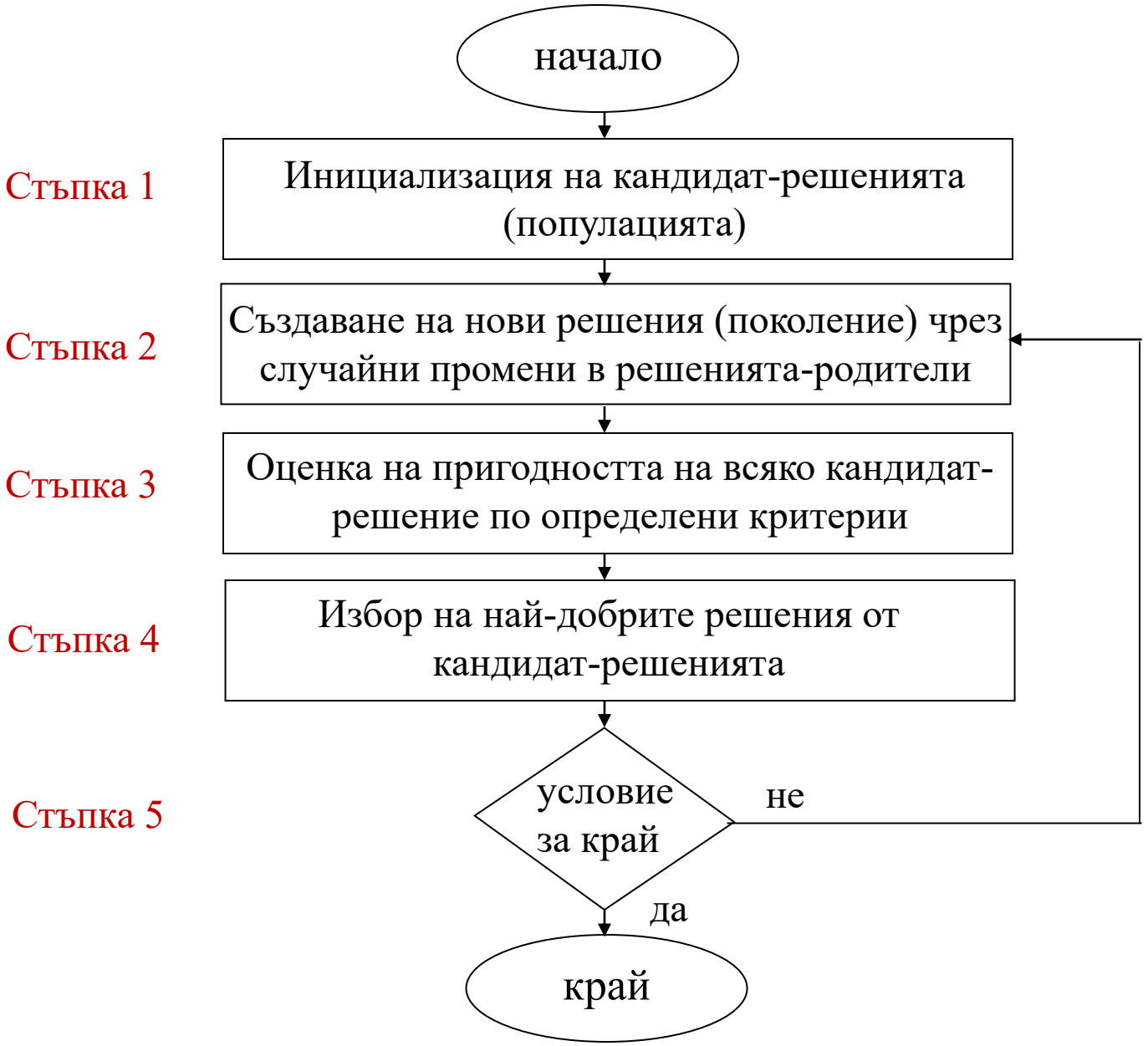
```
RandomizedPartition (a,p,r)
1   i← Randomi(p,r)
2   swap a[p] ↔ a[i]
3   return Partition(a,p,r)

RandomizedQuickSort (a,p,r)
1   if p < r then
2          q← RandomizedPartition(a,p,r)
3          RandomizedQuickSort(a,p,q)
4          RandomizedQuickSort(a,q + 1,r)
```

Генетични алгоритми

Генетичните алгоритми представляват метод за търсене с налучкване, при който основната идея е да се симулират генетичните и еволюционните процеси в природата. Така, за дадена оптимизационна задача, първоначално се построяват няколко произволни неоптимални решения. Най-сполучливите от тях се запазват и на тяхната база се построяват нови с надеждата, че те ще бъдат още по-добри. Неоптималните и безперспективни решения се изолират от по-нататъшно разглеждане. Очевидна е аналогия с процеса на еволюция на видовете в природата (“оцеляване на най-доброто”).

Примерна блокова схема на генетичен алгоритъм



Ще разгледаме как се прилага този метод в добре известната задача за търговския пътник:

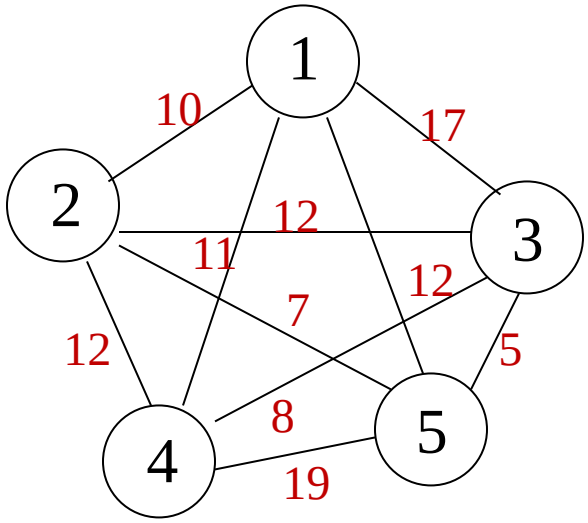
Даден е тегловен граф ***G(V,E)*** със стойности на ребрата реални (положителни и отрицателни) числа. Необходимо е да се намери Хамилтонов цикъл с минимална дължина.

(Задачата е ***NP***-пълна, т. е. се решава с пълно изчерпване на вариантите: за да бъде намерен оптималния цикъл, трябва да се създадат и проверят ***n!*** Хамилтонови цикъла.)

Множеството от кандидатите за решения, които генетичния алгоритъм обработва на всяка стъпка, се нарича *популация*.

В началото построяваме списък $P=\{H_1, H_2, \ldots, H_q\}$ от q произволни Хамилтонови цикли. Те ще представляват началната популация и ще послужат за основа за по-нататъшната "еволюция". Хамилтоновите цикли ще запазваме като пермутация от числата от 1 до n, показваща реда, в който се посещават върховете:

Нека ***n*=5**:



Няколко произволни Хамилтонови цикли в него:

1-2-3-4-5 с цена 25

3-4-5-1-2 с цена 49

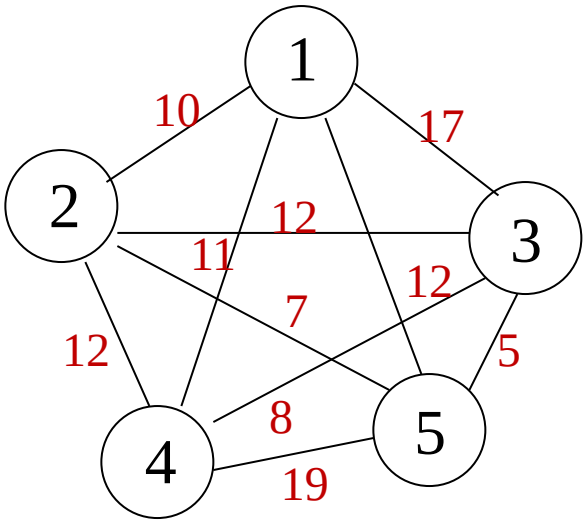
5-3-1-4-2 с цена 35

Процесът на "репродукция" при генетичните алгоритми се извършва по следния начин: на всяка стъпка се избира подмножество на текущата популация (*родители*) и така избраните елементи се комбинират и формират нова група елементи (*наследници* или *поколение*).

Тъй като популацията е ограничена по размер, новополучените елементи ще заменят част от участващите в нея. За целта е необходимо да дефинираме целева функция, на базата на която ще сравняваме елементите и ще отхвърляме "лошите".

В задачата на всяка стъпка ще избираме за родители половината от циклите в ***P*** (тези с най-малката дължина), след което ще ги комбинираме по двойки. Така от всеки два цикъла, принадлежащи на ***P*** - $H_i = (i_1, i_2, \ldots, i_n)$ и $H_j = (j_1, j_2, \ldots, j_n)$ ще се получат два нови: избираме произволни позиции ***k*** и ***r*** ($k<r$) и двата нови цикъла (наследници) ще бъдат: $(i_1, \ldots, i_{k-1}, j_k, \ldots, j_r, i_{r+1}, \ldots, i_n)$ и $(j_1, \ldots, j_{k-1}, i_k, \ldots, i_r, j_{r+1}, \ldots, j_n)$.

Така например, цикълът 2-3-1-4-5, комбиниран с 5-1-3-4-2 за ***k***=2 и ***r***=4 дава като резултат 2-1-3-4-5 (с цена54) и 5-3-1-4-2 (35). Аналогично 1-2-3-4-5, комбиниран с 1-3-5-4-2 за ***k***=2 и ***r***=3 дава 1-3-5-4-5 (?) и 1-2-3-4-2 (?).



От примера се вижда, че в някои случаи се получават повтарящи се (и съответно - липсващи) върхове. В такъв случай повтарящ се елемент се замества с липсващия. Новополучените наследници се добавят към популацията ***P***, след което от нея се изключва "лошата" половина, т. е. цикли с най-голяма дължина. Така, след известен брой стъпки, се очаква алгоритмът да построи Хамилтонов цикъл с дължина, близка до оптималната.

При работа с генетичния алгоритъм се забелязват някои негативни ефекти. Така, например, с многократно прилагане на основната стъпка (комбиниране на най-добрите решения) често се стига до списък ***P*** от почти идентични цикли. В този случай по-нататъшна еволюция няма да доведе до значително подобрение.

Съществуват два начина за решаване на този проблем:
- да се избере по-голяма начална популация (но може да се окаже, че паметта ще е недостатъчна);
- при получаване на два еднакви цикъла да променим единия, разместваики два произволни съседни върха (в генетичните алгоритми това се нарича *мутация*).

Като цяло задачата може да бъде реализирана чрез следните функции:

- initGraph() - за генериране на произволен граф с ***n*** върха;
- randomCycle() - за генериране на произволен Хамилтонов цикъл;
- evaluate() - целева функция, която оценява даден Хамилтонов цикъл (сумира стойностите на ребрата);
- combine() - от два избрани родителя получава два нови наследника;
- mutate() - проверява, дали има съвпадащи елементи в текущата популация и в случай че има, променя единия от тях, като разменя два произволни върха в Хамилтоновия цикъл;
- reproduce() - основна функция. Тя сортира елементи на популацията и определя родителите, както и елементите, които да бъдат отстранени.

Реализацията на алгоритъма - в Преслав Наков, Панайот Добриков. *Програмиране = ++Алгоритми*, TopTeam Co., София, 2002

Генерирането на нови решения (Стъпка 2 или *combine()*) може да става по различни начини.

А) от две пермутации-родители се получават две нови решения;

Б) от две пермутации-родители се получава едно ново решение; например, цикълът 2-3-1-4-5, комбиниран с 5-1-3-4-2 за случайно избрано ***k*=3**, при заместване на елементите в първия с ***k*+1, …, *n*** от втория дава като резултат 2-3-1-4-2. Ако се получи повторение на номер на възел в цикъла, той се замества с липсващия.

В) от всяка пермутация родител се получава нова пермутация (наследник) чрез случаен избор на участък от цикъла, който след това се инвертира; например от 5-1-3-4-2 при ***k*=2** и ***r*=3** се получава 5-4-3-1-2. Този начин съответства на еднополовото размножаване на организмите.

Д.Фогел е предложил, описал, реализирал и експериментално изследвал този алгоритъм за граф със 100 възела, използвайки подхода В) за Стъпка 2. *(С.Стойчев, Синтез и анализ на алгоритми, Издателство "БПС", София, 2003)*