СОРТИРОВКА И ПОИСК ДАННЫХ

1.Алгоритмы сортировки

Сортировку следует понимать как процесс перегруппировки заданного множества объектов в некотором определённом порядке. Цель сортировки — облегчить поиск элементов в таком упорядоченном множестве.
Уточним задачу сортировки.
Пусть надо упорядочить *n* элементов

$R_1, R_2, ..., R_n$,

которые назовём *записями*. Каждая запись R_j имеет свой ключ K_j, который и управляет процессом сортировки. Помимо ключа, запись может содержать дополнительную «сопутствующую информацию», которая не влияет на сортировку, но всегда остаётся в этой записи.
Задача сортировки — найти такую перестановку записей $p(1)\,p(2)...p(n)$, после которой ключи расположились бы в неубывающем порядке

$K_{p(1)} \leqslant K_{p(2)} \leqslant ... \leqslant K_{p(n)}$

При сортировке либо перемещаются сами записи, либо создаётся вспомогательная таблица, которая описывает перестановку и обеспечивает доступ к записям в соответствии с порядком их ключей.
Обычно сортировку подразделяют на два класса: *внутреннюю*, когда все записи хранятся в оперативной памяти, и *внешнюю*, когда они там все не помещаются. Мы будем рассматривать только внутреннюю сортировку.
Основное требование к методам сортировки — экономное использование времени процессора и памяти. Хорошие алгоритмы затрачивают на сортировку *n* записей время порядка $n \log n$.
Существующие методы сортировки обычно разбивают на три класса, в зависимости от лежащего в их основе приёма:
а) сортировка выбором;
б) сортировка обменами;
в) сортировка включениями (вставками).
Рассмотрим основные алгоритмы сортировки на примере сортировки целочисленного массива.
Линейный выбор. Метод предполагает использование рабочего массива, в который помещается отсортированный массив. Количество просмотров определяется количеством элементов массива.
Сортировка посредством линейного выбора сводится к следующему:
1. Найти наименьший элемент, переслать его в рабочий массив и заменить его в исходном массиве величиной, которая больше любого реального элемента.
2. Повторить шаг 1. На этот раз будет выбран наименьший из оставшихся элементов.
3. Повторять шаг 1 до тех пор, пока не будут выбраны все *n* элементов.
Функция, реализующая алгоритм линейного выбора, имеет следующий вид:

```
#include <limits.h>
void lin_wib(int *a, int n)
{ int i, j, imin, amin, *p;
p=(int*)malloc(n*sizeof(int)); // выделяем память под рабочий массив
for(j=0; j<n; j++)
 {
 for(amin=INT_MAX, i=0; i<n; i++)  // ищем минимальный элемент
  if(a[i]<amin) { imin=i; amin=a[i]; }
 p[j]=amin; a[imin]=INT_MAX;
 }
for(j=0;j<n;j++) // отсортированный массив --> на место исходного
 a[j]=p[j];
free(p);
}
```

Линейный выбор с обменом. В этом методе рабочий массив не используется. Общая схема алгоритма следующая.
Просмотрим элементы $a_1, a_2, ..., a_n$, найдём среди них минимальный элемент и переставим его на первое место. Теперь просмотрим элементы $a_2, ..., a_n$, найдём среди них минимальный элемент и поставим его на второе место. И так до тех пор, пока не останется один элемент.

Стандартный обмен (метод "пузырька"). Просматриваем элементы $a_1, ..., a_n$ и попутно меняем местами те соседние элементы, для которых выполнено неравенство $a_i > a_{i+1}$. В результате первого просмотра максимальный элемент станет последним ("он вниз - пузыри вверх"). На следующем просмотре аналогичную процедуру проведем над элементами $a_1, ..., a_{n-1}$ и т. д. Сортировку необходимо закончить, если будет выполнено одно из двух условий: после очередного прохода не сделано ни одного обмена; сделан проход для элементов a_1, a_2.

Челночная сортировка. Челночная сортировка, называемая также "сортировкой просеиванием" или "линейной вставкой с обменом" является наиболее эффективной из всех рассмотренных выше методов и отличается от сортировки обменом тем, что не сохраняет фиксированной последовательности сравнений. Алгоритм челночной сортировки действует точно так же, как стандартный обмен до тех пор, пока не возникает необходимость перестановки элементов исходного массива. Сравниваемый меньший элемент поднимается, насколько это возможно, вверх. Этот элемент сравнивается в обратном порядке со своими предшественниками по направлению к первой позиции. Если он меньше предшественника, то выполняется обмен и начинается очередное сравнение. Когда элемент, передвигаемый вверх, встречает меньший элемент, этот процесс прекращается и нисходящее сравнение возобновляется с той позиции, с которой выполнялся первый обмен.

Сортировка заканчивается, когда нисходящее сравнение выходит на границу массива.

Процесс челночной сортировки можно проследить на следующем примере:

Исходный массив: 2 7 9 5 4

Нисходящие сравнения: 2 7; 7 9; 9 5:

После перестановки: 2 7 5 9 4

Восходящие сравнения и обмен : 7 5 $\rightarrow$ 5 7: 2 5 — конец восходящего сравнения; получен массив 2 5 7 9 4

Нисходящие сравнения: 9 4

После перестановки: 2 5 7 4 9

Восходящие сравнения и обмен : 7 4 $\rightarrow$ 4 7; 5 4 $\rightarrow$ 4 5; 2 4 — конец восходящего сравнения; получен массив 2 4 5 7 9.

Сортировка Шелла. Все рассмотренные выше методы обмена были не столь эффективны при реализации на ЭВМ из-за сравнений и возможных обменов только соседних элементов. Лучших результатов следует ожидать от метода, в котором пересылки выполняются на большие расстояния. Один из таких методов предложил D. L. Shell.

Сортировка Шелла, называемая также "слиянием с обменом", является расширением челночной сортировки.

Идея метода заключается в том, что выбирается интервал *h* между сравниваемыми элементами, который уменьшается после каждого прохода. Для последовательности интервалов выполняются условия:

$h_p = 1, h_{i+1} < h_i, i = 1, 2, ..., p-1$.

Таким образом, вначале сравниваются (и, если нужно, переставляются) далеко стоящие элементы, а на последнем проходе — соседние элементы.

Выигрыш получается за счёт того, что на каждом этапе сортировки либо участвует сравнительно мало элементов, либо эти элементы уже довольно хорошо упорядочены, и требуется небольшое количество перестановок. Последний просмотр сортировки Шелла выполняется по тому же алгоритму, что и в челночной сортировке. Предыдущие просмотры подобны просмотрам челночной обработки, но в них сравниваются не соседние элементы, а элементы, отстоящие на заданное расстояние *h*. Большой шаг на начальных этапах сортировки позволяет уменьшить число вторичных сравнений на более поздних этапах.

При анализе данного алгоритма возникают достаточно сложные математические задачи, многие из которых ещё не решены. Например, неизвестно, какая последовательность расстояний даст лучшие результаты, хотя выяснено, что расстояния h_i не должны быть множителями один другого. Можно рекомендовать следующие последовательности (они записаны в обратном порядке):

а) 1, 4, 13, 40, 121, ...,

где $h_{k-1} = 3h_k + 1, h_p = 1$ и $p = \left[\log_2 n\right] - 1$;

б) 1, 3, 7, 15, 31, ... ,

где $h_{k-1} = 2h_k + 1, h_p = 1$ и $p = \left[\log_2 n\right] - 1$.

Линейная (простая) вставка основана на последовательной вставке элементов в упорядоченный рабочий массив. Линейная вставка чаще всего используется тогда, когда процесс, внешний к данной сортировке, динамически вносит изменения в массив, вес элементы которого известны и который все время должен быть упорядоченным.

Алгоритм линейной вставки следующий. Первый элемент исходного массива помещается в первую позицию рабочего массива. Следующий элемент исходного массива сравнивается с первым. Если этот элемент больше, он помещается во вторую позицию рабочего массива. Если этот элемент меньше, то первый элемент сдвигается на вторую позицию, а новый элемент помещается на первую. Далее все выбираемые из исходного массива элементы последовательно сравниваются с элементами рабочего массива, начиная с первого, до тех пор, пока не встретится больший. Этот больший элемент и все последующие элементы рабочего массива перемещаются на одну позицию вниз, освобождая место для нового элемента.

Центрированная и двоичная вставки. Введение "структуры” в упорядочиваемый рабочий массив в общем случае сокращает количество сравнений, выполняемых при поиске позиции элемента в упорядочиваемом массиве. При использовании "структуры” в сортировке вставкой обычно применяют либо центрированную, либо двоичную вставки. Эти алгоритмы просты, но обладают особенностями реализации, характерными для нелинейных алгоритмов сортировки

Центрированная вставка. Представим рабочий массив состоящим как бы из двух ветвей — нисходящей (левой) и восходящей (правой). Центральный элемент этого массива называется медианой.

Приведём описание алгоритма центрированной вставки.

В позицию, расположенную в середине рабочего массива, помещается первый элемент (он и будет медианой). Нисходящая и восходящая ветви имеют указатели, которые показывают на ближайшие к началу и концу занятые позиции. После загрузки первого элемента в центральную позицию оба указателя совпадают и показывают на него. Следующий элемент исходного массива сравнивается с медианой. Если новый элемент меньше, то он размещается на нисходящей ветви, в противном случае — на восходящей ветви. Кроме того, соответствующий концевой указатель продвигается на единицу вниз (нисходящая ветвь) или единицу вверх (восходящая ветвь).

Каждый последующий элемент исходного массива сравнивается вначале с медианой, а затем с элементами на соответствующей ветви до тех пор, пока не займёт нужную позицию. Если область памяти, выделенная для одной ветви, будет исчерпана, то все элементы рабочего массива сдвигаются в противоположном направлении. Величина сдвига может варьироваться.

Двоичная (бинарная) вставка в отличии от линейной использует для отыскания места вставки алгоритм двоичного (бинарного) поиска, то есть при вставке j-го элемента он сравнивается вначале с элементом $\left[\frac{j}{2}\right]$, затем, если он меньше, сравнивается с элементом $\left[\frac{j}{4}\right]$, а если больше - с $\left[\frac{j}{2}\right]+\left[\frac{j}{4}\right]$ и т. д. до тех пор, пока он не найдёт свое место. Все элементы рабочего массива, начиная с позиции вставки и ниже, сдвигаются на одну позицию вниз, освобождая место для j-го элемента.

Быстрая сортировка (метод Хоара). Метод "пузырька" является не самым эффективным из рассмотренных алгоритмов, поскольку требует большого количества сравнений и обменов. Однако оказалось, что сортировку, основанную на принципе обмена, можно усовершенствовать таким образом, что получится самый хороший из известных на сегодняшний день методов.

Алгоритм быстрой сортировки, предложенный Хоаром, использует тот факт, что для достижения наибольшей эффективности желательно производить обмены элементов на больших расстояниях.

Суть метода заключается в следующем. Найдём такой элемент массива, который разобьёт весь массив на два подмножества: те элементы, которые меньше делящего элемента, и те, которые по величине не меньше его (то есть как бы "отсортируем" один элемент, определив его окончательное местоположение).

Далее применим эту же процедуру к более коротким левому и правому подмножествам.

Таким образом, надо реализовать рекурсивный алгоритм, который сортирует элементы множества, начиная с элемента с индексом left и завершая элементом с индексом right. Условие окончания данного алгоритма — совпадение левой и правой границ подмножества (то есть когда в подмножестве остается один элемент).

Точка деления массива может быть определена следующим образом.

Вводятся два указателя i и j, причём вначале i = left, а j = right. Сравниваются i-й и j-й элементы и, если обмен не требуется, то $j = j - 1$ и этот процесс повторяется. После первого обмена i увеличивается на единицу ($i = i + 1$) и сравнения продолжаются с увеличением i до тех пор, пока не произойдёт еще один обмен. Тогда опять уменьшим j и т. д. пока не станет $i = j$. К моменту, когда $i = j$, элемент a_i займёт свою окончательную позицию, так как слева от него не будет больших элементов, а справа — меньших. Таким образом, поставленная задача решена.

Для повышения эффективности быстрой сортировки можно использовать следующий приём: не применяя рекурсивной процедуры к ставшему коротким подмножеству, для его сортировки перейти к другому методу, например, челночной сортировке. То есть рекурсивная процедура применяется только для массивов, длина которых не менее определённого размера.

9.2. Алгоритмы поиска

Алгоритмы поиска, как и алгоритмы сортировки, являются основными алгоритмами обработки данных как системных, так и прикладных задач.

Дан аргумент поиска *K*. Задача поиска состоит в отыскании записи, имеющей *K* своим ключом.

Существуют две возможности окончания поиска: либо поиск оказался удачным, то есть позволил определить местонахождение записи, содержащей ключ *K*, либо он оказался неудачным, то есть показал, что ни одна из записей не содержит ключ *K*.

Рассмотрим основные алгоритмы поиска. Так как нас интересует в первую очередь сам процесс поиска, а не обнаруженные данные, будем считать, что ключ и запись совпадают и имеют тип **`int`**.

Последовательный (линейный) поиск. Для массива, данные в котором не упорядочены сортировкой или каким-либо другим способом, единственный путь для поиска заданного элемента состоит в сравнении каждого элемента массива с заданным. При совпадении некоторого элемента массива с заданным его позиция в массиве фиксируется. Этот алгоритм называется последовательным или линейным поиском. Эффективность этого метода очень низкая, так как для отыскания одного элемента в массиве размерности *N* в среднем нужно сделать *N*/2 сравнений.

Бинарный (двоичный) поиск. Бинарный или двоичный поиск может быть использован в качестве алгоритма поиска в упорядоченном массиве: $K_1 \leqslant K_2 \leqslant ... \leqslant K_n$.

Схема алгоритма следующая. Сначала сравнивают *K* со средним элементом массива. Результат сравнения позволяет определить, в какой половине массива следует продолжать поиск, применяя к ней ту же процедуру, и т. д. После не более чем $\lfloor \log_2 n \rfloor + 1$ сравнений ключ либо будет найден, либо будет установлено его отсутствие.

Интерполяционный поиск. По-видимому, бинарный поиск не является самым лучшим методом поиска, доказательством чему служит то обстоятельство, что в повседневной жизни им мало пользуются. Гораздо чаще используется интерполяционный поиск. Его схема следующая. Если известно, что *K* находится между K_l и K_r, то номер очередного элемента для сравнения определяется формулой

$$m = l + \frac{(r-1)\cdot(K - K_1)}{K_r - K_l}.$$

Последующая процедура аналогична процедуре, используемой в бинарном поиске.

Интерполяционный поиск асимптотически предпочтительнее бинарного; по существу один шаг бинарного поиска уменьшает количество "подозреваемых" записей от *n* до *n*/2,

а один шаг интерполяционного - от *n* до $\sqrt{n}$. В среднем интерполяционный поиск требует $\log_2 \log_2 n$ шагов.

Примеры решения задач

Задача А

Написать и протестировать функцию сортировки целочисленного массива методом стандартного обмена. Определить время сортировки массива объемом 1000, 5000 и 10000 элементов. Для контроля за сортировкой организовать выдачу 10 элементов из начала, середины и исходного и отсортированного массивов.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <conio.h>
// функция st_obmen() - сортировка методом "пузырька"
void st_obmen(int *a, int n)
{
int i, pp, buf;
do
 { n--;
 for(pp=i=0; i<n; i++)
  if(a[i]>a[i+1])
   { buf=a[i]; a[i]=a[i+1]; a[i+1]=buf; pp=1; }
 }
while(pp);
}
// функция printm() - печать 10 первых, средних и последних элементов массива
void printm(const char *str, int *a, int n)
{ int i, j, in;
printf("\n\n\t\t\t %s \n", str);
for(j=0; j<3; j++,printf("\n"))
 {
 switch(j)
  {
  case 0 : in=0; printf(" начало : "); break;
  case 1 : in=n/2-5; printf("середина : "); break;
  case 2 : in=n-10; printf(" конец : ");
  }
 for(i=in; i<in+10; i++) printf(" %6d",a[i]);
 }
}

int main()
{ int i, j, x[10000], n[3]={1000,5000,10000};
time_t t1, t2, t;
double dt[3];
srand(2213);
for(j=0; j<3; j++)
 {
 for(i=0; i<n[j]; i++) *(x+i)=rand()%n[j];
 if(!j) printm(" Исходный массив (1000 эл.)", x, n[j]);
 t1=time(&t) ;
 st_obmen(x,n[3]);
 t2=time(&t);
 dt[j]=difftime(t2,t1);
 if(!j) printm(" Отсортированный массив", x, n[j]);
 }
printf("\n\n \t Время сортировки ");
for(j=0; j<3; j++)
 printf("\n %5d элементов --> %4.11f c ", n[j], dt[j]);
return 0;
}
```

Задача B

Написать и протестировать функцию поиска ключа в целочисленном массиве методом бинарного поиска.

Тест: поиск m ключей в целочисленном массиве объёма n.

Элементы массива - случайные числа из интервала (0, n - 1); ключи для поиска - случайные числа из интервала (0, n + m - 1).

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
//функция bin_poisk() - бинарный поиск в целочисленном массиве
int bin_poisk(int *a, int n, int v)
{
int l=0, k=n-1, m;
while(l<=k)
 { m=(l+k)/2;
 if(v<a[m]) k=m-1;
 else
  if(v>a[m]) l=m+1;
  else return m; // нашли!
 }
return -1; // не нашли...
}
// функция st_obmen () - сортировка методом "пузырька"
void st_obmen(int *a, int n)
{
int i, pp, buf;
do
 { n--;
 for(pp=i=0; i<n; i++)
  if(a[i]>a[i+1])
   { buf=a[i]; a[i]=a[i+1]; a[i+1]=buf; pp=1; }
 }
while(pp);
}
int main()
{ int i, n, m, *a, k, pk;
srand(time(NULL));
for(;;)
 { printf(" Введите размер массива (для выхода - 0): "); scanf("%d",&n) ;
 if(n<=0) break;
 a=(int*)malloc(n*sizeof(int));
 printf (" Введите число разыскиваемых ключей : ");
 scanf ("%d",&m);
 for(i=0; i<n; i++) *(a+i)=rand()%n;
 st_obmen(a, n);
 printf("\n\t Аргумент \t Позиция ");
 for(i=0; i<m; i++)
  {
  k=rand()%(n+m);
  pk=bin_poisk(a, n, k);
  printf ("\n\t %6d" , k);
  if(pk==-1) printf("\t\t не найден ");
  else printf("\t\t %5d",pk);
  }
 printf("\n");
 }
}
```

Сортировка и поиск

Задача 1. Провести сравнительный анализ эффективности методов сортировки вставками: линейной, двоичной, центрированной.
Предлагаемый тест: сортировка целочисленного массива размера *n*, элементы которого — случайные величины, распределённые в интервале $(0, n-1)$.

Задача 2. Исследовать эффективность методов поиска: последовательного, бинарного, интерполяционного.
Предлагаемый тест: поиск *m* элементов в целочисленном массиве длины *n*.
Элементы массива — случайные величины, распределённые в интервале $(0, n-1)$. Ключи поиска — случайные величины, распределённые в интервале $(0, n+m-1)$.

Задача 3. Провести сравнительный анализ эффективности следующих методов сортировки:
1) линейный выбор с обменом, челночная сортировка, двоичная вставка;
2) сортировка Шелла, центрированная вставка;
3) стандартный обмен, быстрая сортировка, линейная вставка.
Предлагаемый тест: сортировка целочисленного массива размера *n*, элементы которого — случайные величины, распределённые в интервале $(0, n-1)$.

Задача 4. Написать и протестировать функции сортировки целочисленных массивов и поиска ключей в них по следующим методам:
1) челночная сортировка, бинарный поиск;
2) сортировка Шелла, бинарный поиск (рекурсивная функция);
3) быстрая сортировка, бинарный поиск;
4) центрированная вставка, интерполяционный поиск.
Тест сортировки: сортировка целочисленного массива размера *n*, элементы которого — случайные величины, распределённые в интервале $(0, n-1)$.

Тест поиска: поиск *m* элементов в отсортированном массиве.

Задача 5. Написать и протестировать функции сортировки записей и поиска их по ключам для следующих методов:
1) линейный выбор с обменом, бинарный поиск;
2) челночная сортировка, бинарный поиск;
3) сортировка Шелла, бинарный поиск;
4) быстрая сортировка, бинарный поиск;
5) стандартный обмен, интерполяционный поиск;
6) линейная вставка, интерполяционный поиск;
7) центрированная вставка, бинарный поиск.
Запись имеет три поля, например, фамилия, имя, номер телефона. Иметь не менее 30 записей. Поиск — по любому ключу, задаваемому из меню.

Задача 6. Провести сравнительный анализ эффективности не менее двух вариантов быстрой сортировки целочисленных массивов размера $n > 5000$.

Задача 7. Написать и протестировать функцию челночной сортировки целочисленного массива с управляемым направлением (возрастание/убывание) сортировки по признаку.
Заголовок функции должен иметь вид **`void shatl(int *a, int n, char *pr)`**,
`a` — сортируемый массив;
`n` — размер массива;
`pr` — признак, управляющий направлением сортировки: **`pr="incr"`** — сортировка по возрастанию; **`pr="decr"`** — сортировка по убыванию.

Задача 8. Исследовать возможности адаптации различных методов сортировки к структуре исходного массива. С этой целью определить время сортировки целочисленного массива объёма *n* для следующих вариантов представления исходного массива:
неупорядоченный,
почти упорядоченный,
упорядоченный в противоположном направлении.
Методы, подлежащие исследованию:
1) линейный выбор с обменом, центрированная вставка;
2) челночная сортировка, линейная вставка;
3) сортировка Шелла, линейная вставка;
4) быстрая сортировка, бинарная вставка;
5) стандартный обмен, бинарная вставка.

РЕШЕНИЕ СИСТЕМ ЛИНЕЙНЫХ АЛГЕБРАИЧЕСКИХ УРАВНЕНИЙ

Предлагается разработать и протестировать функции, поддерживающие матричные операции и решение систем линейных алгебраических уравнений, оценить эффективность созданных программ.
Характеристики эффективности: время работы; точность.
При вычислении матрицы, обратной матрице $A_{n,n}$, точность оценивается с помощью модифицированной *k*-нормы матрицы **D**, равной
$\mathbf{D} = \mathbf{A} \cdot \mathbf{A}^{-1} - \mathbf{E}$,
где **E** — единичная матрица.
Модифицированная *k*-норма
При решении системы линейных алгебраических уравнений; *n*-го порядка характеристикой точности является *невязка*, равная

$$\delta = \frac{1}{n}\sqrt{\sum_{i=1}^{n}\left(b_i - \bar{b}_i\right)^2},$$

где $\bar{\mathbf{b}} = \mathbf{A}\bar{\mathbf{x}}$, **x** — решение системы линейных уравнений $\mathbf{Ax} = \mathbf{b}$.
Невязка — величина ошибки (расхождения) приближённого равенства.

Примеры решении задач

Задача А
Написать и протестировать функцию для обращения положительно определённых симметрических матриц методом Гаусса.
Пусть **A** — действительная матрица порядка *n*×*n* и уравнение
$\mathbf{y} = \mathbf{Ax}$ (14.1)
определяет некоторое линейное преобразование. Если $a_{11} \neq 0$, то можно разрешить первое из уравнений системы (14.1) относительно x_1 и подставить это выражение в оставшиеся уравнения. Это приведёт к системе

$$x_1 = a'_{11}y_1 + a'_{12}x_2 + \ldots + a'_{1n}x_n;$$
$$y_2 = a'_{21}y_1 + a'_{22}x_2 + \ldots + a'_{2n}x_n;$$
$$\ldots$$
$$y_n = a'_{n1}y_1 + a'_{n2}x_2 + \ldots + a'_{nn}x_n,$$

где $a'_{11} = \frac{1}{a_{11}}$, $a'_{1j} = -\frac{a_{1j}}{a_{11}}$, $a'_{i1} = \frac{a_{i1}}{a_{11}}$, $a'_{ij} = a_{ij} - \frac{a_{i1}a_{1j}}{a_{11}}$, $i, j = 2, \ldots, n$.
На следующем шаге исключают переменную x_2 при условии, что $a'_{22} \neq 0$. Если возможно выполнить *n* таких шагов, то возникает система
$\mathbf{x} = \mathbf{By}$, (14.3)
где $\mathbf{B} = \mathbf{A}^{-1}$, то есть выполнено обращение исходной матрицы. Если на *k*-м шаге матрицы коэффициентов представить в виде блоков

$$\begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix}, \quad (14.4)$$

то блок $\mathbf{A}_{11}$ размера *k*×*k* определяет матрицу, обратную соответствующему блоку матрицы **A**, в то время как $\mathbf{A}_{22}$ есть матрица, обратная соответствующему блоку матрицы $\mathbf{A}^{-1}$. Всё это достаточно просто установить, если положить $x_{k+1} = \ldots = x_n = 0$ в первых *k* уравнениях вида (14.1) и соответственно $y_1 = \ldots = y_k = 0$ в последних $n - k$ уравнениях вида (14.3).
Если матрица **A** положительно определённая, то при любом, *k* матрицы $\mathbf{A}_{11}$ и $\mathbf{A}_{22}$ также положительно определённые. Следовательно, первый диагональный элемент матрицы $\mathbf{A}_{22}$ положителен, и, таким образом, для положительно определённых матриц следующий шаг всегда возможен. Кроме того, для симметрических матриц нет необходимости вычислять элементы, расположенные выше главной диагонали.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#define N 8
#define MEM (float*)malloc(n*n*sizeof(float))
#define RND (1.0-(float)rand()/32768.0*2)
FILE *fp;
// Функция umn_m() - перемножение матриц
void umn_m(float *a,float *b,float *c,int n,int m,int p)
{ int i,j,k; float s;
for(i=0;i<n;i++)
 for(j=0;j<p;j++)
  { for(s=k=0;k<m;k++) s+=*(a+i*m+k)**(b+k*p+j);
  *(c+i*p+j)=s;
  }
}
// Функция print_m() - вывод матриц (результаты сбрасываются в файл)
void print_m(const char *str, float *a, int n, int m)
{ int i,j;
fprintf(fp,"\n\t\t %s \n",str);
for(i=0;i<n;fprintf(fp,"\n"),i++)
 for(j=0;j<m;j++)
  fprintf(fp,"%10.3e ",*(a+i*m+j));
}
// Функция gauss() - обращение матриц методом Гаусса
// Обратная матрица получается на месте исходной!
int gauss(float *a,int n)
{ int i,j,k; float *h,p,q;
// Выделяем память под вспомогательный вектор h(n)
h=(float*)malloc(n*sizeof(float));
for(k=n-1;k>=0;k--)
 { p=*a;
 if(p<=0) return 1;
 for(i=1; i<n; i++)
  { q=*(a+i*n);
  h[i]=((i>k)?q:-q)/p;
  for(j=1;j<=i;j++)
   *(a+(i-1)*n+j-1)=*(a+i*n+j)+q*h[j];
  }
 *(a+n*n-1)=1/p;
 for(i=1;i<n;i++)
  for(j=0;j<i;j++)
   *(a+(n-1)*n+i-1)=h[i];
 }
for(i=1;i<n;i++)
 for(j=0;j<i;j++)
  *(a+j*n+i)=*(a+i*n+j);
free(h);
return 0;
}
```

```
int main()
{ int i,j,n; float *a,*a1,*c,r;
// Открываем файл gauss.txt для записи результатов
if((fp=fopen("gauss.txt","w"))==NULL)
 printf("\n Ошибка при открытии файла "),exit(0) ;
printf("\n Введите размерность матрицы n<8 : ");
scanf("%d",&n);
// Выделяем память под матрицы A, A1, C
a=MEM; a1=MEM; c=MEM;
//Формируем и выводим исходную
//симметрическую положительно определённую матрицу
srand(time(NULL));
for(i=0;i<n;i++)
 for(j=0;j<=i;j++)
  {
  if(i==j) r=2*n*fabs(RND);// a(i,i)>0; a(i,i)>>a(i,j)
  else r=RND;
  *(a1+i*n+j)=*(a1+j*n+i)=*(a+i*n+j)=*(a+j*n+i)=r;
  }
print_m("Исходная матрица A1",a,n,n);
// Обращаем матрицу A и выводим её
if(gauss(a1,n))
 fprintf(fp,"\nАварийное прекращение работы фун. gauss() "),exit(1) ;
print_m("Обратная матрица A1",a1,n,n);
// Проверяем: C=A*A1, выводим результат
umn_m(a,a1,c,n,n,n); print_m("Матрица C=A*A1",c,n,n);
// Освобождаем память, закрываем файл gauss.txt
free(a); free(a1); free(c); fclose(fp); return 0;
}
```

Решение систем линейных уравнений

Задача 1. Написать и протестировать функции, реализующие следующие матричные операции:
1) сложение матриц;
2) умножение матрицы на скаляр;
3) транспонирование матриц;
4) умножение матриц;
5) перестановка двух строк матрицы;
5) перестановка столбцов матрицы согласно вектору транспозиции.

Задача 2. Написать и протестировать функции, реализующие следующие матричные операции:
1) вычитание матриц;
2) изменение знака матрицы;
3) формирование единичной матрицы;
4) умножение матрицы слева на её транспонированную;
5) перестановка двух столбцов матрицы;
6) перестановка строк матрицы согласно вектору транспозиции.

Задача 3. Сформировать матрицу $A_{n,m}$, элементами которой являются случайные числа, равномерно распределённые в интервале $(-5{,}4;\ 8{,}2)$. Вычислить следующие нормы матрицы:

1) $\|A\|_m = \max\limits_i \sum\limits_j |a_{ij}|$ (m-норма)

2) $\|A\|_l = \max\limits_j \sum\limits_i |a_{ij}|$ (l-норма);

3) $\|A\|_k = \sqrt{\sum\limits_{i=1}^{n}\sum\limits_{j=1}^{m} a_{ij}^2}$ (k-норма).

Задача 4. Сформировать положительно определенную симметрическую матрицу и вычислить для неё m-, l- и k-нормы (см. задачу 3).

Задача 5. Написать и протестировать функцию, вычисляющую определитель матрицы по формуле

$$\det A = \sum_{(p_1,\,p_2,\,\ldots,\,p_n)} (-1)^k\, a_{1,p_1} a_{2,p_2} \ldots a_{n,p_n},$$

где сумма распространяется на всевозможные перестановки $(p_1, p_2, \ldots, p_n)$ элементов $1, 2, \ldots, n$, причём $k = 0$, если перестановка чётная и $k = 1$, если перестановка нечётная.

Задача 6. Написать и протестировать функцию вычисления обратной матрицы методом окаймления (деления на клетки).
Исследовать зависимость точности и времени обращения от типа представления (**`float`**, **`double`**) и размера матриц.
Описание метода
Пусть имеем неособенную матрицу *M* размера *n*×*n*. Разобьём её на четыре клетки

$$\mathbf{M} = \begin{bmatrix} \mathbf{a} & \mathbf{b} \\ \mathbf{c} & \mathbf{d} \end{bmatrix},$$

где *a* — (*p*×*p*)-подматрица; *b* — (*p*×*q*)-подматрица; *c* — (*q*×*p*)-подматрица; *d* — (*q*×*q*)-подматрица $(n = p + q)$.

Матрица $\mathbf{M}^{-1}$ существует и может быть разбита на клетки тем же самым образом, что и матрица **M**, то есть

$$\mathbf{M}^{-1} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}$$

где **A**, **B**, **C**, **D** — соответственно (*p*×*p*)-, (*p*×*q*)-, (*q*×*p*)-, (*q*×*q*)-подматрицы.
Пусть подматрица **d** имеет обратную и что $\mathbf{d}^{-1}$ известна. Тогда, поскольку $\mathbf{M} \cdot \mathbf{M}^{-1} = \mathbf{E}$, имеем

$$\begin{bmatrix} \mathbf{a} & \mathbf{b} \\ \mathbf{c} & \mathbf{d} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} = \begin{bmatrix} \mathbf{E}_p & \mathbf{0} \\ \mathbf{0} & \mathbf{E}_q \end{bmatrix}$$

или

$$\mathbf{aA} + \mathbf{bC} = \mathbf{E}_p;\ \mathbf{aB} + \mathbf{bD} = \mathbf{0};\ \mathbf{cA} + \mathbf{dC} = \mathbf{0};\ \mathbf{cB} + \mathbf{dD} = \mathbf{E}_q.$$

Так как обратная матрица $\mathbf{d}^{-1}$ известна, то после небольших преобразований получим формулы, которые могут быть последовательно решены относительно матриц **A**, **B**, **C** и **D**:

$$\mathbf{A} = \left(\mathbf{a} - \mathbf{b}\mathbf{d}^{-1}\mathbf{c}\right)^{-1};\ \mathbf{B} = -\mathbf{A}\mathbf{b}\mathbf{d}^{-1};\ \mathbf{C} = -\mathbf{d}^{-1}\mathbf{c}\mathbf{A};\ \mathbf{D} = \mathbf{d}^{-1} - \mathbf{d}^{-1}\mathbf{c}\mathbf{B}.$$

Поскольку матрица $\mathbf{M}^{-1}$ существует, существуют и матрицы **A**, **B**, **C** и **D**. Следовательно, если матрица $\mathbf{d}^{-1}$ известна, могут быть вычислены и матрицы **A**, **B**, **C** и **D**.
Технически это можно организовать с помощью метода *окаймления*. Сущность его заключается в следующем.
Пусть дана матрица

$$M = \begin{bmatrix} m_{11} & \dots & m_{1n} \\ \dots & \dots & \dots \\ m_{n1} & \dots & m_{nn} \end{bmatrix}$$

Образуем последовательность матриц

$$M_1 = \begin{bmatrix} m_{11} \end{bmatrix};$$

$$M_2 = \begin{bmatrix} \mathbf{M}_1 & m_{12} \\ m_{21} & m_{22} \end{bmatrix};$$

$$\mathbf{M}_3 = \begin{bmatrix} & & m_{13} \\ & \mathbf{M}_2 & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix}.$$

и т. д. Каждая следующая матрица получена из предыдущей при помощи окаймления. Обратная к первой из этих матриц находится непосредственно:

$$\mathbf{M}_1^{-1} = \frac{1}{m_{11}}.$$

Зная матрицу $\mathbf{M}_1^{-1}$ и применив к $\mathbf{M}_2$ приведённую выше схему вычисления, можно получить $\mathbf{M}_2^{-1}$, а затем при помощи $\mathbf{M}_2^{-1}$ аналогично получить $\mathbf{M}_3^{-1}$ и т. д. Наконец, матрица $\mathbf{M}_n^{-1} = \mathbf{M}^{-1}$. Этот же процесс может быть начат и с правого нижнего угла матрицы **M**.
Задача 7. Написать и протестировать функцию вычисления определителя матрицы по методу Гаусса.

Задача 8. Написать и протестировать функцию вычисления обратной матрицы методом Ершова. Исследовать зависимость точности и времени обращения от типа представления (**`float`**, **`double`**) и размера матриц.

Обращение матриц методом Ершова (методом пополнения) производится следующим образом. На основе исходной матрицы **A** и единичной матрицы **E** по индукции строится последовательность матриц

$\mathbf{A}^{(0)}, \mathbf{A}^{(1)'}, \mathbf{A}^{(1)}, \mathbf{A}^{(2)'}, ..., \mathbf{A}^{(n)'}, \mathbf{A}^{(n)}$, где $\mathbf{A}^{(0)} = \mathbf{A} - \mathbf{E}$,

$$\mathbf{a}_{ij}^{(m)'} = \begin{cases} \mathbf{a}_{ij}^{(m-1)}, \text{ если } i \neq m; \\ 1, \text{ если } i = m \text{ и } i = j; \\ 0, \text{ если } i = m \text{ и } i \neq j. \end{cases}$$

$$\mathbf{a}_{ij}^{(m)} = \mathbf{a}_{ij}^{(m)'} - \frac{\mathbf{a}_{im}^{(m)'} \mathbf{a}_{mj}^{(m-1)}}{1 + \mathbf{a}_{mm}^{(m-1)}}; i, j, m = 1, 2, ..., n.$$

Матрица $\mathbf{A}^{(n)} = \mathbf{A}^{-1}$. Матрицы $\mathbf{A}^{(1)'}, \mathbf{A}^{(2)'}, ..., \mathbf{A}^{(n)'}$ являются вспомогательными.

Задача 9. Написать, протестировать и оценить эффективность функции вычисления обратной матрицы методом Фаддеева. Исследовать зависимость точности и времени обращения от типа представления (**`float`**, **`double`**) и размера матриц.

Вычисление обратной матрицы $\mathbf{A}^{-1}$ порядка *n* по методу Фаддеева производится по следующим формулам:

$$\mathbf{A}_1 = \mathbf{A}, \qquad p_1 = \mathrm{Sp}\mathbf{A}_1, \qquad \mathbf{B}_1 = \mathbf{A}_1 - p_1\mathbf{E};$$

$$\mathbf{A}_1 = \mathbf{A}\mathbf{B}_1, \qquad p_2 = \frac{1}{2}\mathrm{Sp}\mathbf{A}_2, \qquad \mathbf{B}_2 = \mathbf{A}_2 - p_2\mathbf{E};$$

$$... \qquad ... \qquad ...$$

$$\mathbf{A}_{n-1} = \mathbf{A}\mathbf{B}_{n-2}, \quad p_{n-1} = \frac{1}{n-1}\mathrm{Sp}\mathbf{A}_{n-1}, \quad \mathbf{B}_{n-1} = \mathbf{A}_{n-1} - p_{n-1}\mathbf{E};$$

$$\mathbf{A}_n = \mathbf{A}\mathbf{B}_{n-1}, \qquad p_n = \frac{1}{n}\mathrm{Sp}\mathbf{A}_n, \qquad \mathbf{A}^{-1} = \frac{1}{p_n}\mathbf{B}_{n-1}.$$

Здесь $\mathrm{Sp}\mathbf{A} = \sum_{i=1}^{n} a_{ii}$ — след матрицы **A**.

Задача 10. Написать и протестировать функцию, выполняющую разложение неособенной квадратной матрицы на произведение двух треугольных. Может быть использован следующий алгоритм.

Пусть **A** — действительная неособенная матрица порядка *n*. В общем случае матрицу **A** можно разложить в произведение вида

$\mathbf{A} = \mathbf{L} \cdot \mathbf{U}$,

где **L** — нижняя треугольная матрица с единичной диагональю;

U — верхняя треугольная матрица.

Элементы матриц **L** и **U** вычисляются следующим образом:

$$\mathbf{a}_{i,p} = \mathbf{A}_{i,p} - \sum_{k=1}^{p-1} \mathbf{L}_{i,k}\mathbf{U}_{k,p};$$

$$\mathbf{L}_{i,p} = \frac{\mathbf{a}_{i,p}}{\mathbf{a}_{p,p}}, i = p, ..., n;$$

$$\mathbf{U}_{p,p} = \mathbf{a}_{p,p};$$

$$\mathbf{U}_{p,j} = \mathbf{A}_{p,j} - \sum_{k=1}^{p-1} \mathbf{L}_{p,k}\mathbf{U}_{k,j}, j = p+1, ..., n;$$

$p = 1, 2, ..., n$.

Разложение нельзя выполнить, если какая-либо главная подматрица матрицы **A** — особенная. Предусмотреть защиту от возможного деления на ноль.

Задача 11. Написать и протестировать функцию, выполняющую разложение неособенной квадратной матрицы на произведение нижней треугольной матрицы с единичной диагональю и матрицы с ортогональными строками.

Может быть использован следующий алгоритм. Пусть дана действительная неособенная матрица

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}.$$

Из каждой *i*-ой строки матрицы **A**, начиная со второй, вычтем первую строку, умноженную на некоторое число λ_{i1} $(i = 2, \dots, n)$, зависящее от номера строки. В результате будем иметь преобразованную матрицу $\mathbf{A}^{(1)}$.

Множители λ_{i1} должны быть такими, чтобы первая строка матрицы $\mathbf{A}^{(1)}$ была ортогональна всем остальным строкам, то есть

$$\lambda_{i1} = \frac{\sum_{j=1}^{n} a_{1j} a_{ij}}{\sum_{j=1}^{n} a_{1j}^2}.$$

Над матрицей $\mathbf{A}^{(1)}$ проделываем аналогичную операцию: из каждой её *i*-й строки $(i = 3, \dots, n)$ вычтем вторую строку матрицы $\mathbf{A}^{(1)}$, умноженную на λ_{i2}

$$\lambda_{i2} = \frac{\sum_{j=1}^{n} a_{2j} a_{ij}}{\sum_{j=1}^{n} \left(a_{2j}^{(1)}\right)^2}.$$

Получим матрицу $\mathbf{A}^{(2)}$ и т. д., пока не получится матрица $\mathbf{A}^{(n-1)}$ все строки которой попарно ортогональны. Матрица $\mathbf{A}^{(n-1)} = \mathbf{R}$ с ортогональными строками получилась из матрицы **A** в результате цепи элементарных преобразований. Поэтому справедливо равенство $\mathbf{A} = \mathbf{\Lambda}^{-1}\mathbf{R}$, где $\mathbf{\Lambda}$ — нижняя треугольная матрица. Матрицу $\mathbf{\Lambda}$ легко восстановить, проделав над единичной матрицей **E** все элементарные преобразования, совершенные над матрицей **A**. Окончательно имеем $\mathbf{A} = \mathbf{\Lambda}^{-1}\mathbf{R}$.

Задача 12. Вычислить значение матричного выражения $\left(\left(\mathbf{A}+\mathbf{B}\right)^{\mathrm{T}} + \mathbf{E} - \mathbf{A}^{\mathrm{T}} - \mathbf{B}^{\mathrm{T}}\right)^{-1}$.

Матрицы **A** и **B** — общего вида размерности *n*×*n*.

Задача 13. Вычислить значение матричного выражения

$$\left(\mathbf{ABC}\right)^{-1} + \mathbf{E} - \mathbf{C}^{-1}\mathbf{B}^{-1}\mathbf{A}^{-1},$$

где **A**, **B**, **C** — матрицы общего вида размерности *n*×*n*.

Задача 14. Вычислить значение матричного выражения $\mathbf{PL}^{\mathrm{T}}\mathbf{C}^{-1}\mathbf{LP}$, где $\mathbf{C} = \mathbf{LL}^{\mathrm{T}}$.

Матрица **P** — симметричная размерности *k*×*k*, матрица **L** — общего вида размерности *n*×*k*.

Задача 15. Вычислить значение матричного выражения

$$\left(\mathbf{AB} - \mathbf{BA}\right)^{-1},$$

где **A** и **B** — квадратные матрицы размерности *n* с элементами:

$$a_{ij} = \begin{cases} 1 \text{ при } j = i+1, \\ 0 \text{ при } j \neq i+1; \end{cases}$$

$$b_{ij} = \begin{cases} d\left(i+1\right) \text{ при } i = j+1, \\ 0 \text{ при } i \neq j+1; \end{cases}$$

Задача 16. Вычислить значение матричного выражения

$$\left(\left(\mathbf{AB}\right)^{\mathrm{T}} + \mathbf{E} - \mathbf{B}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\right)^{-1},$$

где **A**, **B** — матрицы общего вида размерности *n*×*n*.

Задача 17. Написать, протестировать и оценить эффективность функции решения системы линейных алгебраических уравнений методом Гаусса.
Исследовать зависимость точности и времени решения от типа представления коэффициентов уравнений (**`float`**, **`double`**) и порядка системы.
Задача 18. Написать, протестировать и оценить эффективность функции решения системы линейных алгебраических уравнений методом Гаусса с выбором главного элемента.
Исследовать зависимость точности и времени решения от типа представления коэффициентов уравнений (**`float`**, **`double`**) и порядка системы.
Задача 19. Написать, протестировать и оценить эффективность функции решения системы линейных алгебраических уравнений методом Гаусса-Жордана.
Исследовать зависимость точности и времени решения от типа представления коэффициентов уравнений (**`float`**, **`double`**) и порядка системы.

Задача 20. Написать, протестировать и оценить эффективность функции решения системы линейных алгебраических уравнений методом ортогонализации строк.
Исследовать зависимость точности и времени решения от типа представления коэффициентов уравнений (**`float`**, **`double`**) и порядка системы.
Описание метода. Пусть дана система линейных уравнений
$\mathbf{Ax} = \mathbf{b}\left(\det \mathbf{A} \neq 0\right)$.
Преобразуем строки системы так, чтобы матрица **A** перешла в матрицу **R** с ортогональными строками. При этом вектор **b** перейдет в вектор $\boldsymbol{\beta}$. В результате получим эквивалентную систему $\mathbf{Rx} = \boldsymbol{\beta}$, откуда $\mathbf{x} = \mathbf{R}^{-1}\boldsymbol{\beta}$.
Из свойств ортогональных матриц следует:
$\mathbf{R}^{-1} = \mathbf{R}^{\mathrm{T}}\left(\mathbf{R}\mathbf{R}^{\mathrm{T}}\right)^{-1}$,
где $\mathbf{R}\mathbf{R}^{\mathrm{T}} = \mathbf{D}$; **D** — диагональная матрица.
Поэтому имеем
$\mathbf{x} = \mathbf{R}^{T}\mathbf{D}^{-1}\boldsymbol{\beta}$.
Матрица, обратная диагональной, ищется просто

$$\mathbf{D} = \begin{pmatrix} d_{11} & 0 & \ldots & 0 \\ 0 & d_{22} & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & \ldots & d_{nn} \end{pmatrix} \Rightarrow \mathbf{D}^{-1} = \begin{pmatrix} d_{11}^{-1} & 0 & \ldots & 0 \\ 0 & d_{22}^{-1} & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & \ldots & d_{nn}^{-1} \end{pmatrix}.$$

Матрица **R** может быть получена следующим образом.
Из каждой *i*-й строки системы $\left(i = 3, \ldots, n\right)$ вычтем первую строку, умноженную на λ_{i1}. Получим матрицу $\mathbf{A}^{(1)}$.
Множители λ_{i1} должны быть такими, чтобы первая строка матрицы $\mathbf{A}^{(1)}$ была ортогональна всем остальным строкам, то есть

$$\lambda_{i1} = \frac{\sum_{j=1}^{n} a_{1j}a_{ij}}{\sum_{j=1}^{n} a_{1j}^{2}}.$$

Над матрицей $\mathbf{A}^{(1)}$ проделываем аналогичную операцию: из каждой её *i*-й строки $\left(i = 3, \ldots, n\right)$ вычтем вторую строку матрицы $\mathbf{A}^{(1)}$, умноженную на λ_{i2}

$$\lambda_{i2} = \frac{\sum_{j=1}^{n} a_{2j}a_{ij}}{\sum_{j=1}^{n}\left(a_{2j}^{(1)}\right)^{2}}.$$

Получим матрицу $\mathbf{A}^{(2)}$ и т. д., пока не получится матрица $\mathbf{A}^{(n-1)}$, все строки которой попарно ортогональны.
Систему $\mathbf{Ax} = \mathbf{b}$ можно решить и по-другому. Пусть она приведена к виду $\mathbf{Rx} = \boldsymbol{\beta}$, как это описано выше.
Умножим каждое уравнение системы на

$$\mu_i = \frac{1}{\sqrt{\sum_{j=1}^{n} r_{ij}^{2}}},\ i = 1, \ldots, n.$$

Получим
$\tilde{\mathbf{R}}\mathbf{x} = \tilde{\boldsymbol{\beta}}$
где $\tilde{\mathbf{R}}$ — ортогональная матрица. Поскольку у ортогональных матриц транспонированная матрица совпадает с обратной, то
$\mathbf{x} = \tilde{\mathbf{R}}^{-1}\tilde{\boldsymbol{\beta}} = \tilde{\mathbf{R}}^{\mathrm{T}}\tilde{\boldsymbol{\beta}}$.

Задача 21. Написать и протестировать функцию, выполняющую разложение неособенной симметрической матрицы **A** на произведение двух треугольных, транспонированных между собой матриц $\mathbf{A} = \mathbf{T}'\mathbf{T}$, где

$$\mathbf{T} = \begin{pmatrix} t_{11} & t_{12} & \dots & t_{1n} \\ 0 & t_{22} & \dots & t_{2n} \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & t_{nn} \end{pmatrix}, \mathbf{T}' = \begin{pmatrix} t_{11} & 0 & \dots & 0 \\ t_{12} & t_{22} & \dots & 0 \\ \dots & \dots & \dots & \dots \\ t_{1n} & t_{2n} & \dots & t_{nn} \end{pmatrix}.$$

Описание алгоритма. Перемножив **T** и **T'**, получим уравнения для определения t_{ij}:

$t_{1i}t_{1j} + t_{2i}t_{2j} + \dots + t_{ii}t_{ij} = a_{ij}, (i < j)$;

$t_{1i}^2 + t_{2i}^2 + \dots + t_{ii}^2 = a_{ii}$.

Отсюда

$$t_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} t_{ki}^2}, (1 \leqslant i \leqslant n);$$

$t_{ij} = 0, (i > j)$.

Коэффициенты матрицы **T** будут действительными, если $t_{ij}^2 > 0, i = 1, \dots, n$.

Задача 22. Написать, протестировать и оценить эффективность функции решения системы линейных алгебраических уравнений с ленточными матрицами.

Исследовать зависимость точности и времени решения от типа представления коэффициентов уравнений (**`float`**, **`double`**) и порядка системы.

Описание метода. Если **A** — положительно определённая ленточная матрица такая, что

$a_{ij} = 0, |i - j| > m$,

то существует действительная невырожденная нижняя треугольная матрица **L**, допускающая представление исходной матрицы в виде

$\mathbf{L}\mathbf{L}^{\mathrm{T}} = \mathbf{A}$,

где $l_{ij} = 0$, если $i - j > m$.

Элементы матрицы **L** можно определить по строкам, приравнивая элементы в обеих частях последнего уравнения. Если принять, что все элементы l_{pq} при $q \leqslant 0$ и $q > p$ равны нулю, то элементы *i*-ой строки удовлетворяют соотношениям

$$l_{ij} = \frac{a_{ij} - \sum_{k=i-m}^{i-1} l_{ik}l_{jk}}{l_{jj}}, j = i - m, \dots, i - 1;$$

$$l_{ii} = \sqrt{a_{ii} - \sum_{k=i-m}^{i-1} l_{ik}^2}.$$

Решение системы уравнений $\mathbf{A}\mathbf{x} = \mathbf{b}$ осуществляется в два этапа

$\mathbf{L}\mathbf{y} = \mathbf{b}$;

$\mathbf{L}^{\mathrm{T}}\mathbf{x} = \mathbf{y}$.

Учитывая ширину ленточной матрицы, получаем следующий алгоритм для решения системы уравнений:

$$y_i = \frac{b_i - \sum_{k=i-m}^{i-1} l_{ik}y_k}{l_{ii}}, i = 1, \dots, n;$$

$$x_i = \frac{y_i - \sum_{k=i+1}^{i+m} l_{ki}x_k}{l_{ii}}, i = n, \dots, 1.$$

Задача 23. Написать, протестировать и оценить эффективность функции решения системы линейных алгебраических уравнений по схеме Холецкого. Исследовать зависимость точности и времени решения от типа представления коэффициентов уравнений (**`float`**, **`double`**) и порядка системы.

Описание метода. Дана линейная система $\mathbf{Ax} = \mathbf{b}$, где $\mathbf{A} = \left[a_{ij} \right]$ — симметрическая положительно определённая матрица, для которой справедливо разложение $\mathbf{A} = \mathbf{LDL}^{\mathrm{T}}$,
где **L** — нижняя треугольная матрица с единичной диагональю; **D** — положительно определённая диагональная матрица.
Такое разложение может быть выполнено за *n* шагов, причём на *i*-м шаге определяют *i*-ую строку матрицы **L** и *i*-й элемент d_i матрицы **D**. Выражения для нахождения этих элементов имеют вид:

$$l_{ij} d_j = a_{ij} - \sum_{k=1}^{i-1} l_{ik} d_k l_{jk},\ j = 1, ..., i-1;$$

$$d_i = a_{ii} - \sum_{k=1}^{i-1} l_{ik} d_k l_{ik}\ .$$

После того, как матрицы **L** и **D** будут найдены, заменим исходную систему двумя эквивалентными ей системами

$\mathbf{Ly} = \mathbf{b}$;

$\mathbf{L}^{\mathrm{T}}\mathbf{x} = \mathbf{D}^{-1}\mathbf{y}$.

Эти уравнения можно решить, последовательно вычисляя величины

$$y_i = b_i - \sum_{k=1}^{i-1} l_{ik} y_k,\ i = 1, ..., n;$$

$$x_i = \frac{y_i}{d_i} - \sum_{k=i+1}^{n} l_{ki} x_k,\ i = n, ..., 1.$$

Задача 24. Написать, протестировать и оценить эффективность функции решения системы линейных алгебраических уравнений методом квадратного корня. Исследовать зависимость точности и времени решения от типа представления коэффициентов уравнений (**`float`**, **`double`**) и порядка системы.

Описание метода. Дана линейная система

$\mathbf{Ax} = \mathbf{b}$,

где $\mathbf{A} = \left[a_{ij} \right]$ — симметрическая положительно определённая матрица.

Представим матрицу **A** в виде произведения двух треугольных транспонированных между собой матриц (см. задачу 21)

$\mathbf{A} = \mathbf{T}'\mathbf{T}$.

Система имеет единственное решение, если $t_{ii} \neq 0, i = 1, ..., n$.

После того, как матрица **T** будет найдена, заменим исходную систему двумя эквивалентными ей системами с треугольными матрицами

$\mathbf{T}'\mathbf{y} = \mathbf{b}$,

$\mathbf{Tx} = \mathbf{y}$.

Отсюда можно последовательно найти

$$y_i = \frac{b_i - \sum_{k=1}^{i-1} t_{ki} y_k}{t_{ii}},\ i = 1, 2, ..., n;$$

$$x_i = \frac{y_i - \sum_{k=i+1}^{n} t_{ik} x_k}{t_{ii}},\ i = n, n-1, ..., 1.$$

Задача 25. Написать и протестировать функцию вычисления определителя симметрической положительно определённой матрицы путём её разложения на две взаимно транспонированные треугольные матрицы.
Описание метода. Если **A** — симметрическая положительно определённая матрица, то существует действительная невырожденная нижняя треугольная матрица **L**, такая, что

$\mathbf{L}\mathbf{L}^{\mathrm{T}} = \mathbf{A}$.

Элементы матрицы **L** можно определять по строкам или столбцам, приравнивая соответствующие элементы матриц в приведённом выражении. Если матрицу вычисляют по строкам, то для элементов *i*-й строки справедливы следующие соотношения:

$$l_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2}\ ;$$

$$l_{ij} = \frac{a_{ij} - \sum_{k=1}^{i-1} l_{ik} l_{jk}}{l_{jj}},\ j = 1, ..., i-1\,.$$

Определитель матрицы A можно вычислить в соответствии с выражением

$$\det \mathbf{A} = \det \mathbf{L} \cdot \det \mathbf{L}^{\mathrm{T}} = \left(\prod_{i=1}^{n} l_{ii} \right)^2 .$$