

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ  
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «ДОНСКОЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

**(ДГТУ)**

Факультет «Информатика и вычислительная техника»

Кафедра «Программное обеспечение вычислительной техники и автоматизированных систем»

**Лабораторная работа №5**

По дисциплине: «Эвристические методы и алгоритмы»

Выполнил студент группы ВПР32

Бояршинов Никита

Проверил

проф. Кобак В.Г.

Ростов-на-Дону  
2024

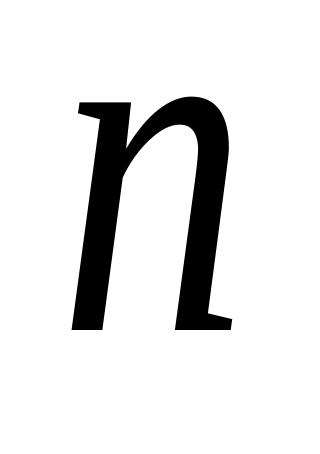
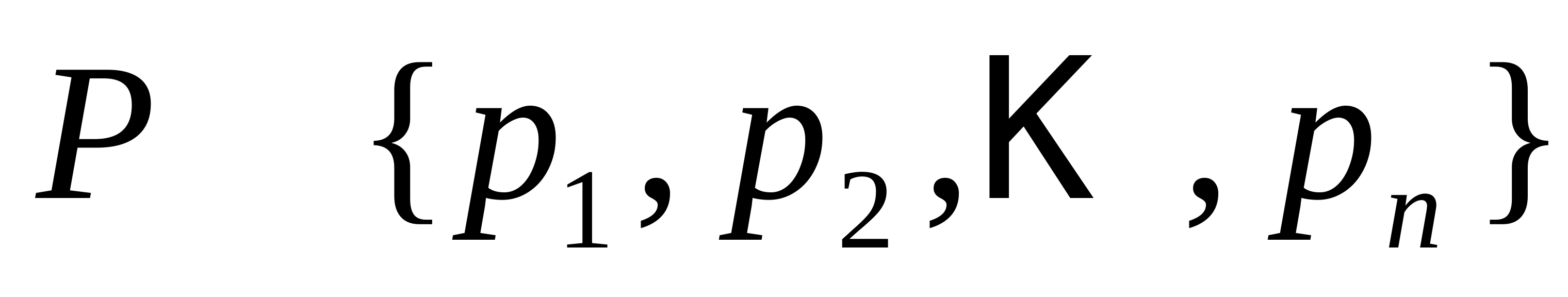
### **Введение**

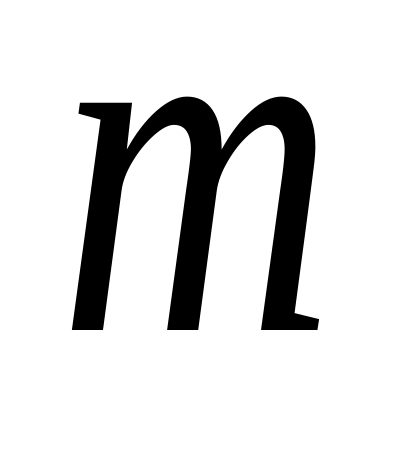
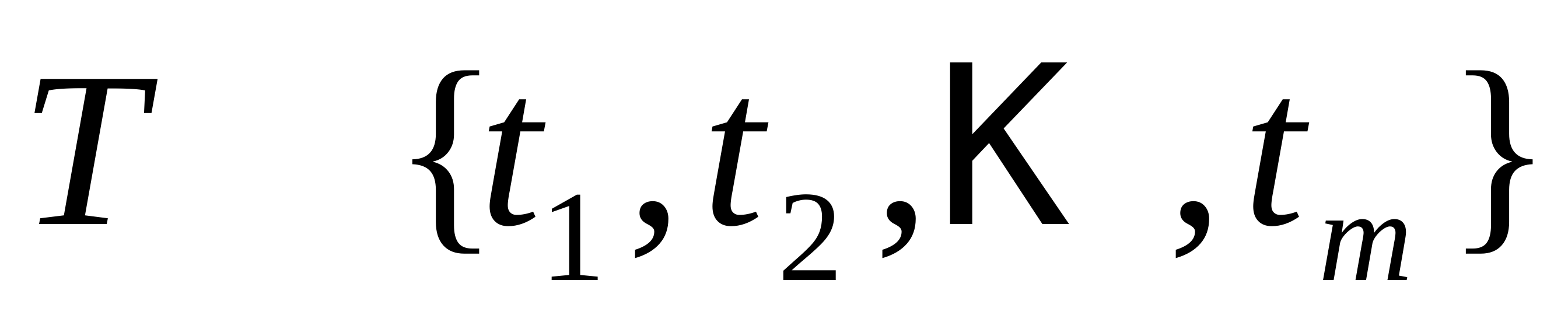
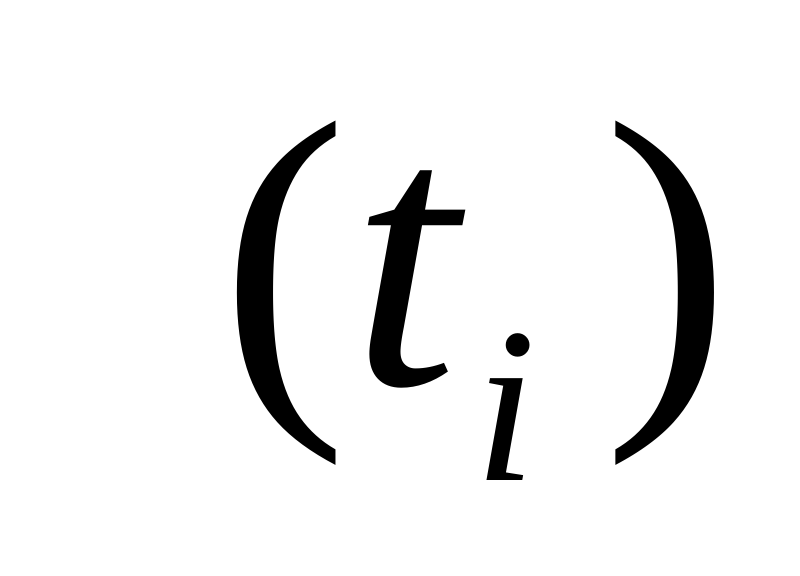
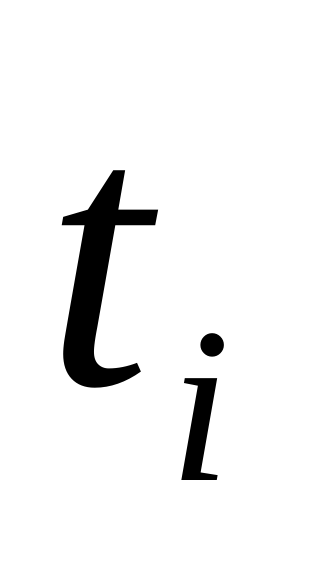
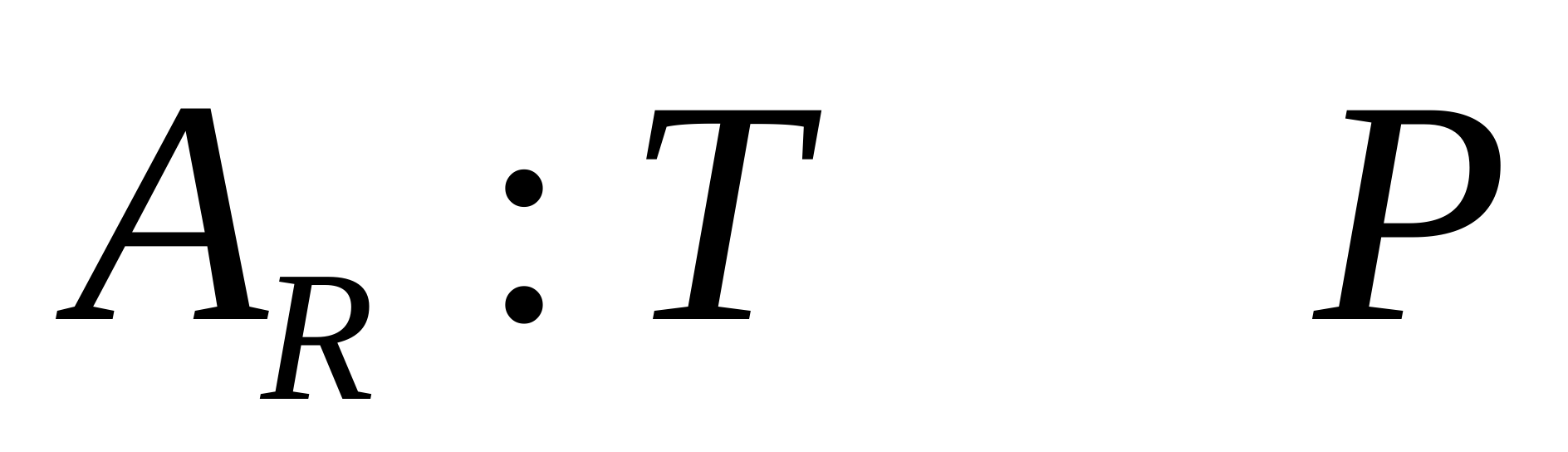
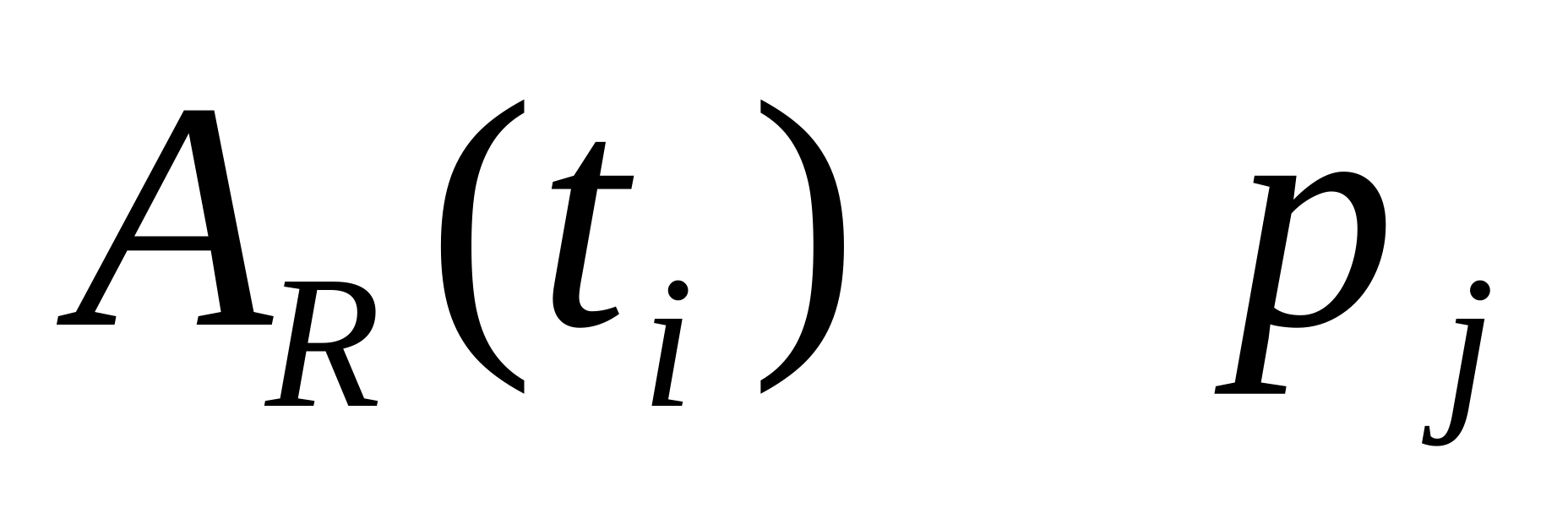
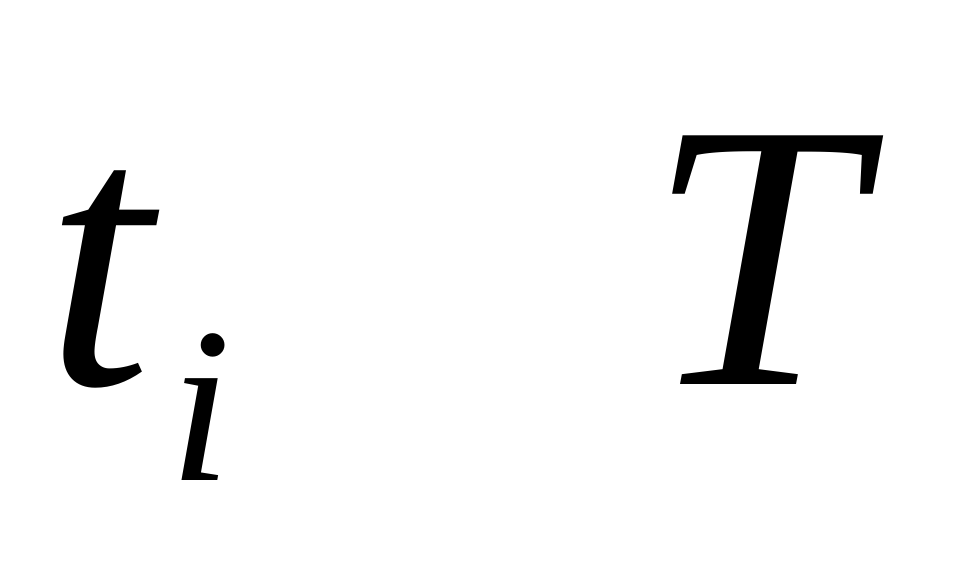
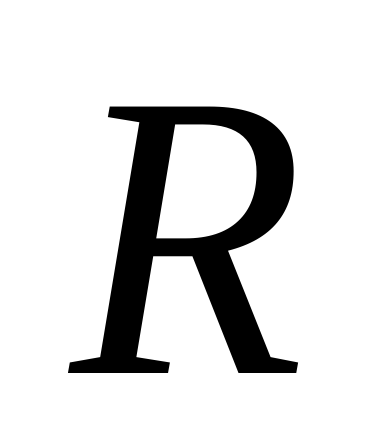
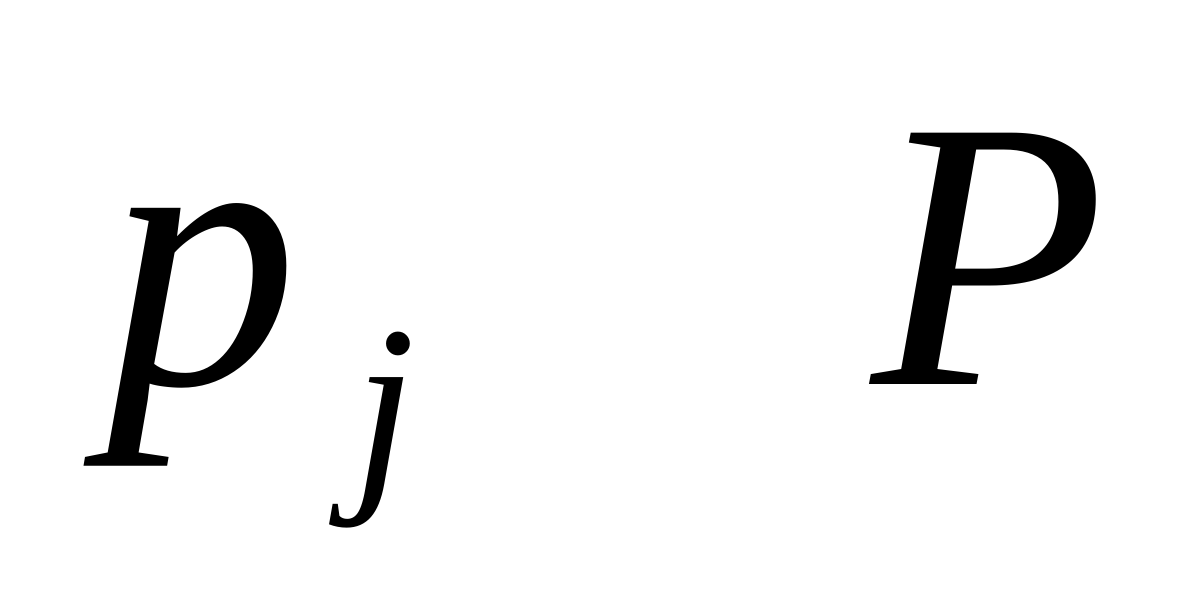
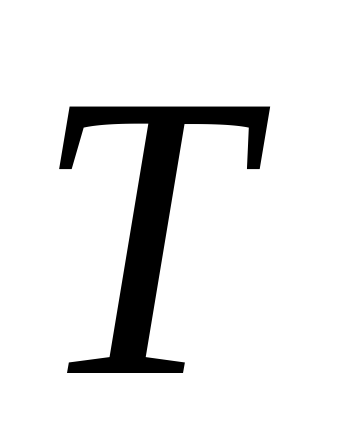
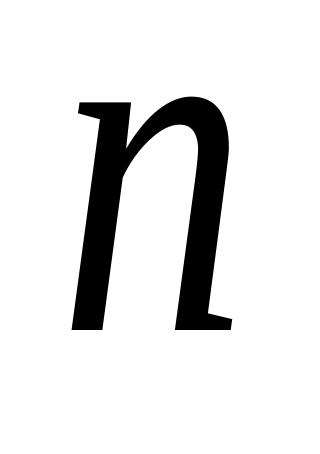
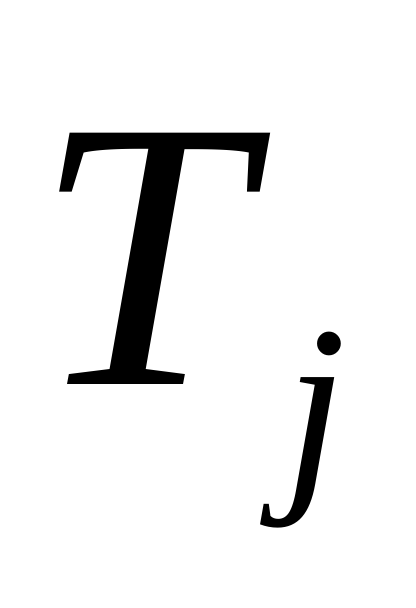
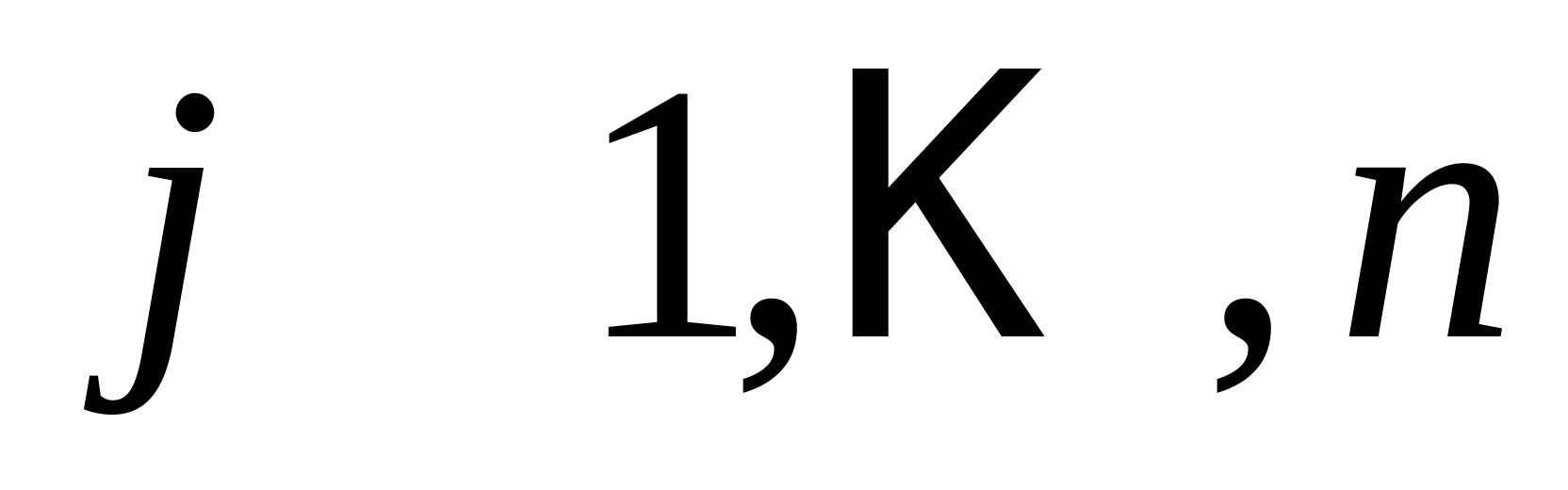
Предметом области исследования расписаний является круг задач проектирования и организационного управления в различных системах, в которых требуется найти наилучшее (оптимальное) значение выбранных критериев их функционирования с учетом имеющихся ограничений.

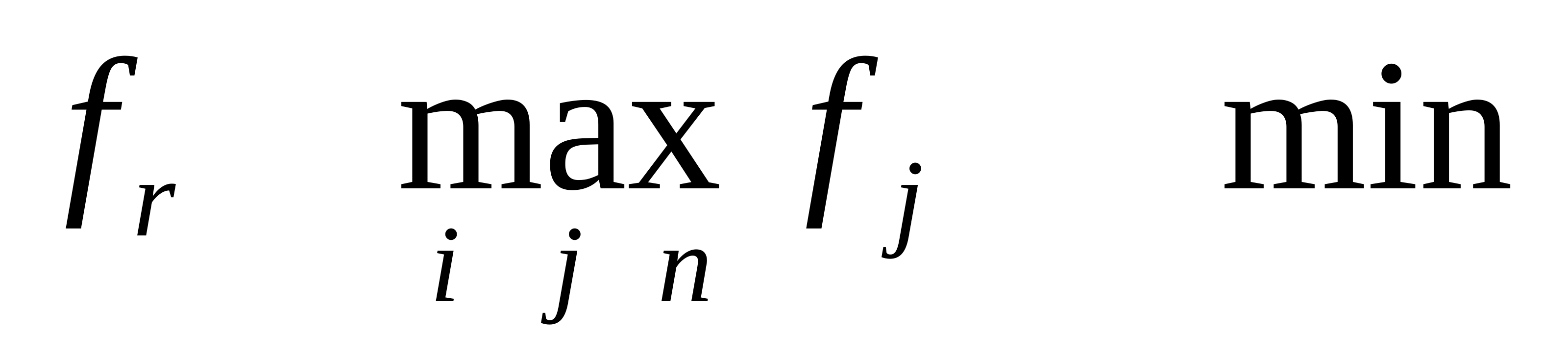
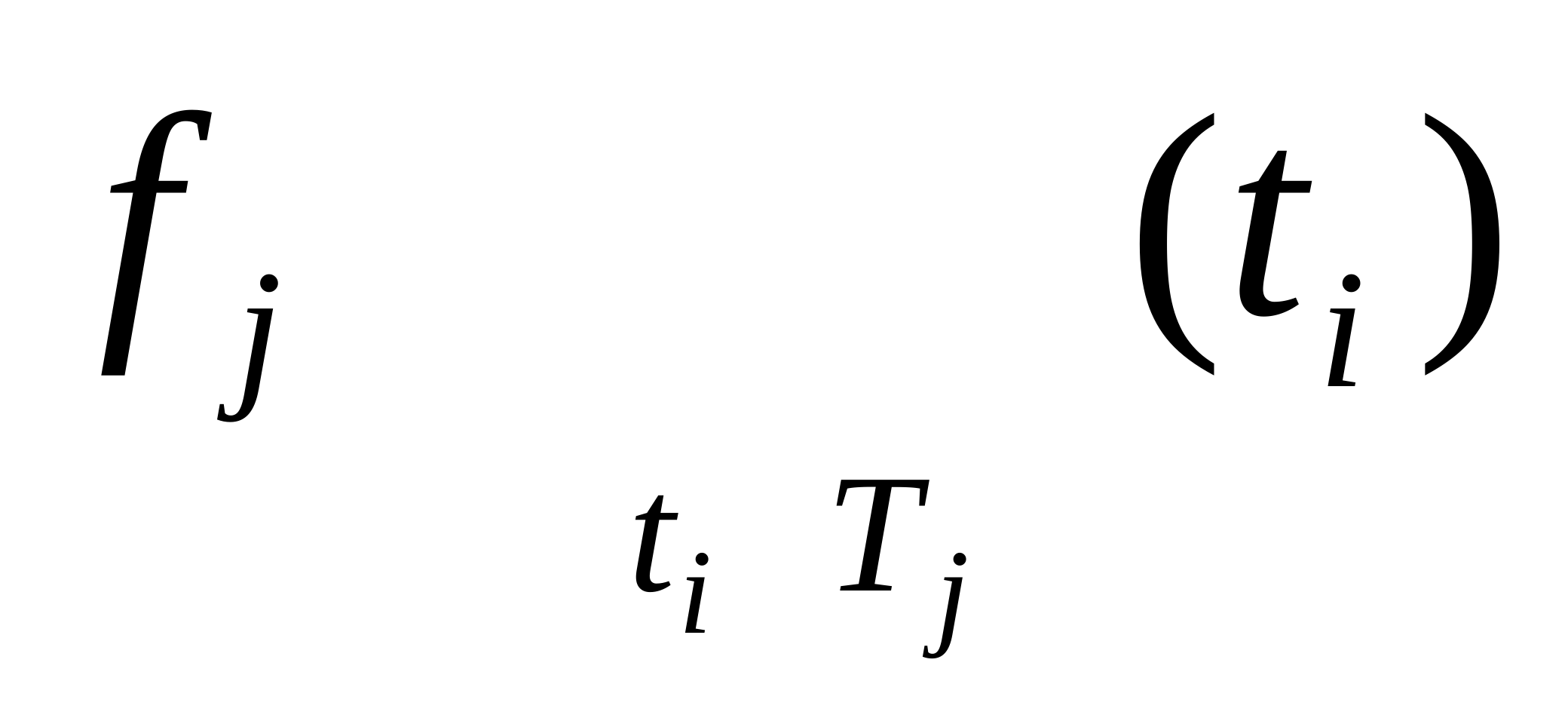
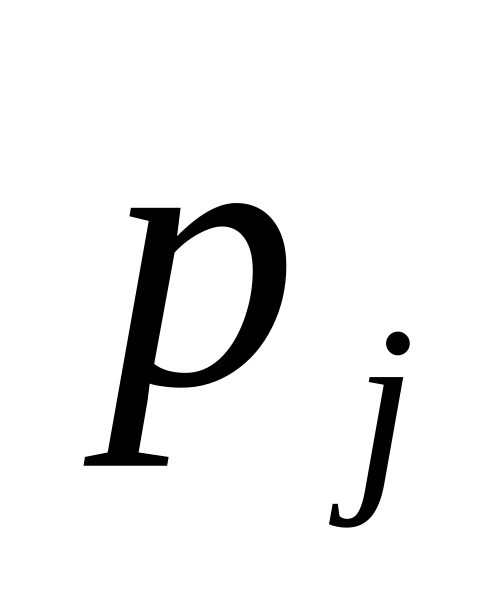
Программирование для многопроцессорных машинных систем связано с распараллеливанием и синхронизацией вычислений и организацией выполнения параллельных вычислительных процессов. Это выдвигает целый ряд сложных задач, среди которых весьма важными являются расчет характеристик времени и количества операций, требующихся для выполнения параллельных программ и построения расписаний (планов), выполнения параллельных программ на многопроцессорных и многомашинных вычислительных системах.

Модели параллельных программ и операционные характеристики процессов их выполнения служат основой для планирования параллельных вычислительных процессов, т.е. для построения расписаний указанных процессов. Расписания параллельных вычислительных процессов определяют порядок выполнения программы на вычислительной системе, включая распределение частей программы по процессам. С увеличением числа распределяемых частей программ и количества используемых процессоров сложность построения оптимальных расписаний обычно резко возрастает. Поэтому важное значение имеют простые в построении и удобные в реализации приближенные расписания параллельных вычислительных процессов, близкие к оптимальным расписаниям с точки зрения времени выполнения параллельных программ.

### **Постановка задачи**

Имеется вычислительная система (ВС), состоящая из  несвязанных идентичных устройств (приборов, процессоров и т.п.) .

На обслуживание в ВС поступает набор из  независимых параллельных заданий (работ) , известно время решения  задания  на любом из устройств. При этом каждое задание может выполняться на любом из устройств (процессоре), в каждый момент времени отдельный процессор обслуживает не более одного задания и выполнение задания не прерывается для передачи на другой процессор. Требуется найти такое распределение заданий по процессорам, при котором суммарное время выполнения заданий на каждом из процессоров было бы минимальным. Под расписанием следует понимать отображение , такое, что если , то говорят, что задание  в расписании , назначенного на процессор . При сделанных выше допущениях расписание можно представить разбиением множества заданий на  непересекающихся подмножеств , где .

Критерий, используемый для минимизации времени завершения обслуживания заданий, является минимальным критерием и определяется в следующем виде: , где  - время завершения работы процессора .

### **Модифицированная модель Голдберга для решения однородной минимаксной задачи**

Генетические алгоритмы (ГА) являются одной из парадигм эволюционных вычислений, представляют собой алгоритмы поиска лучшего, а не оптимального решения задачи, построены на принципах, сходных с принципами естественного отбора и генетики.

ГА имеет вероятностную природу и в связи с этим результаты, получаемые с помощью него отличаются в каждом запуске и определяются случайной последовательностью, переданной в схему алгоритма. Точность алгоритма зависит не только от входной последовательности случайных чисел, но и от условий задачи, таких как размерность задачи и конкретное распределение весов.

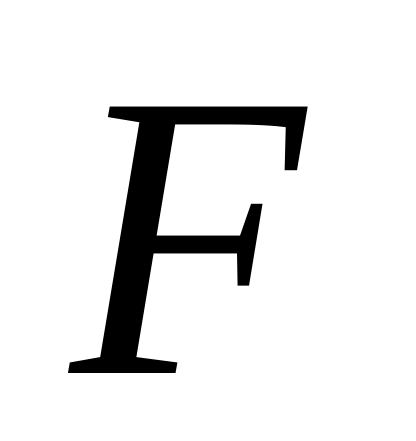
Рассмотрим общую схему работы генетического алгоритма:

Ш.1 Формируется начальное поколение, состоящее из заданного числа особей.

Ш.2 Отбор особей и применение ГА операторов кроссовера и мутации с заданной вероятностью для создания нового поколения.

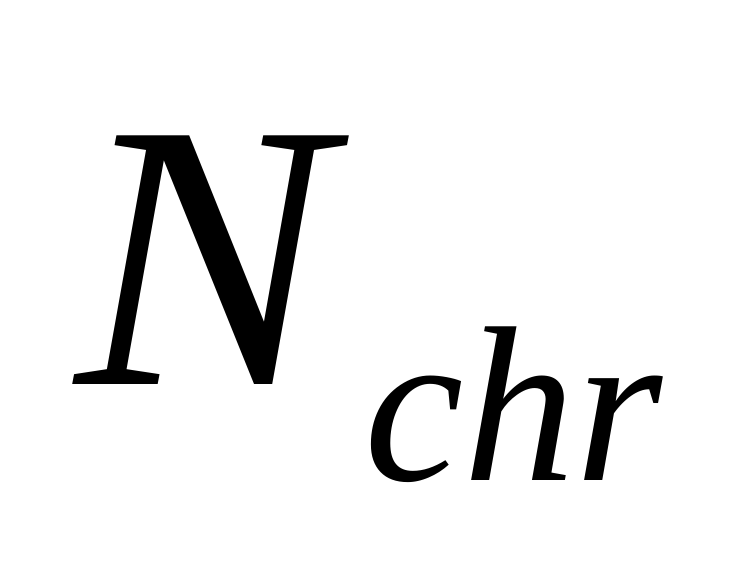
Ш.3 Проверка условия останова, которая обычно заключается в неизменности лучшего решения в течение заданного числа поколений. Если проверка прошла не успешно, то переход на Ш.2.

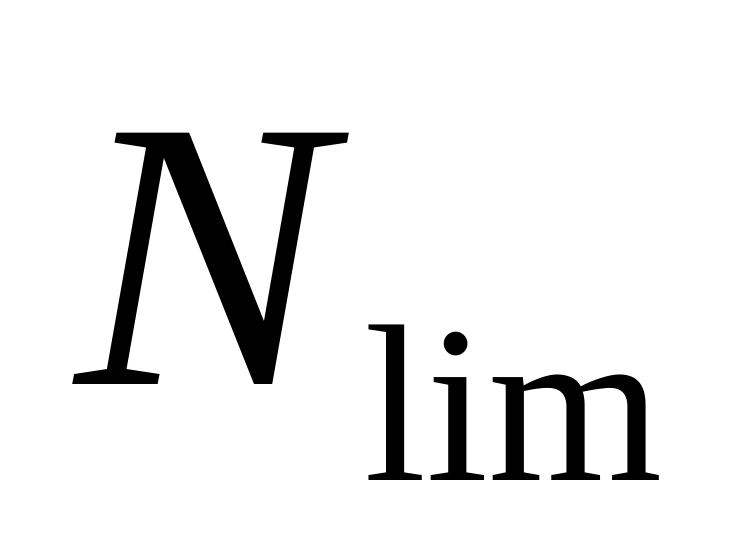
Ш.4 Лучшая особь выбирается как найденное решение.

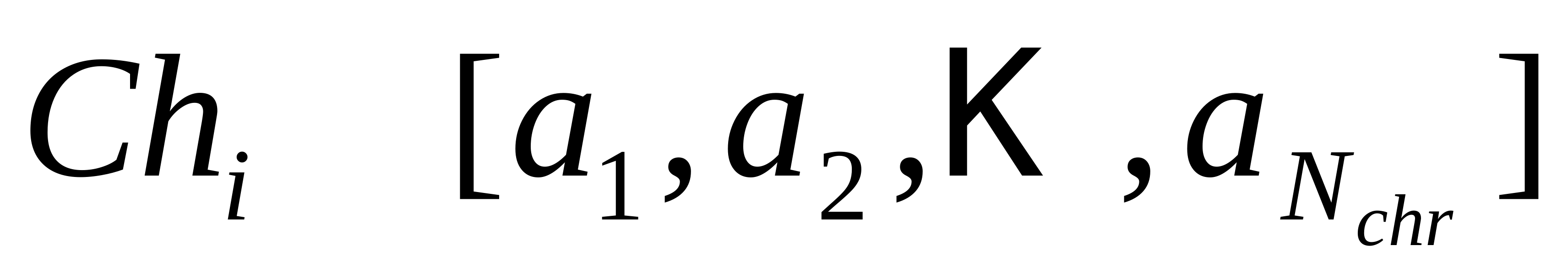
ГА является общим алгоритмом для решения любой задачи, и при его применении к конкретной необходимо выбрать механизм кодирования параметров задачи (фенотипа) в гены особи (генотипа), определить оптимизационную функцию  (fitness function) и выбрать условия останова.

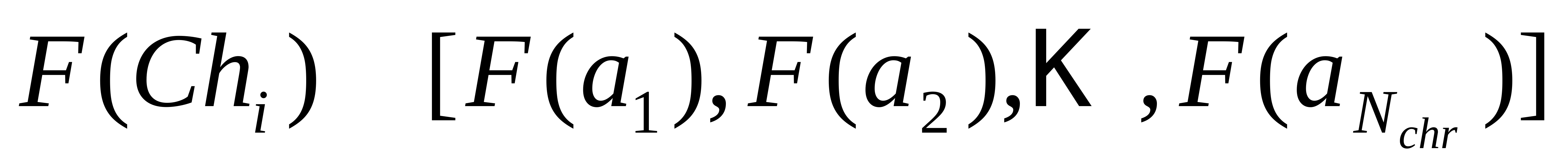
В данном случае решения задачи теории расписания минимаксный критерий будет являться оптимизационной функцией, а условием останова - неизменность лучшего решения в течение заданного числа поколений.

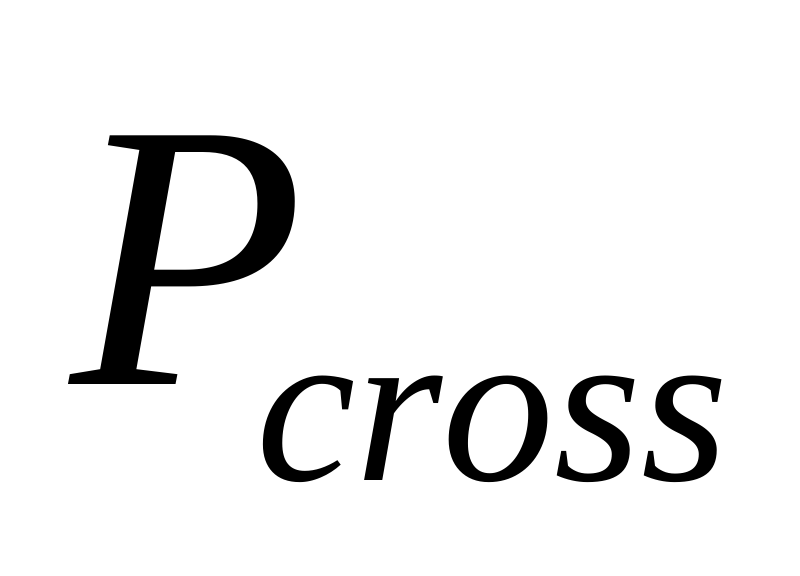
Введем следующие обозначения:

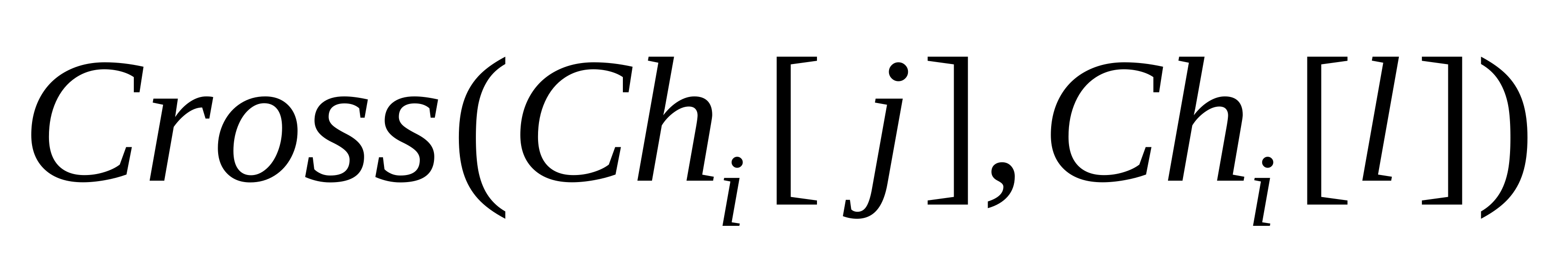
 - число особей (хромосом);

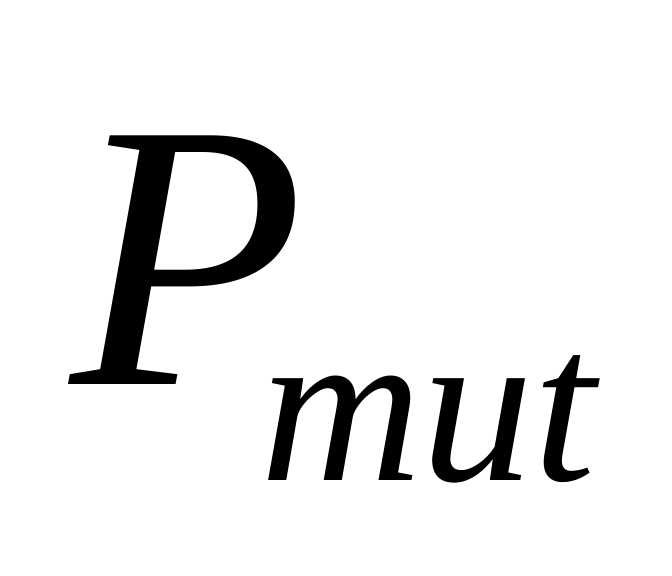
 - условие останова по количеству одинаковых поколений;

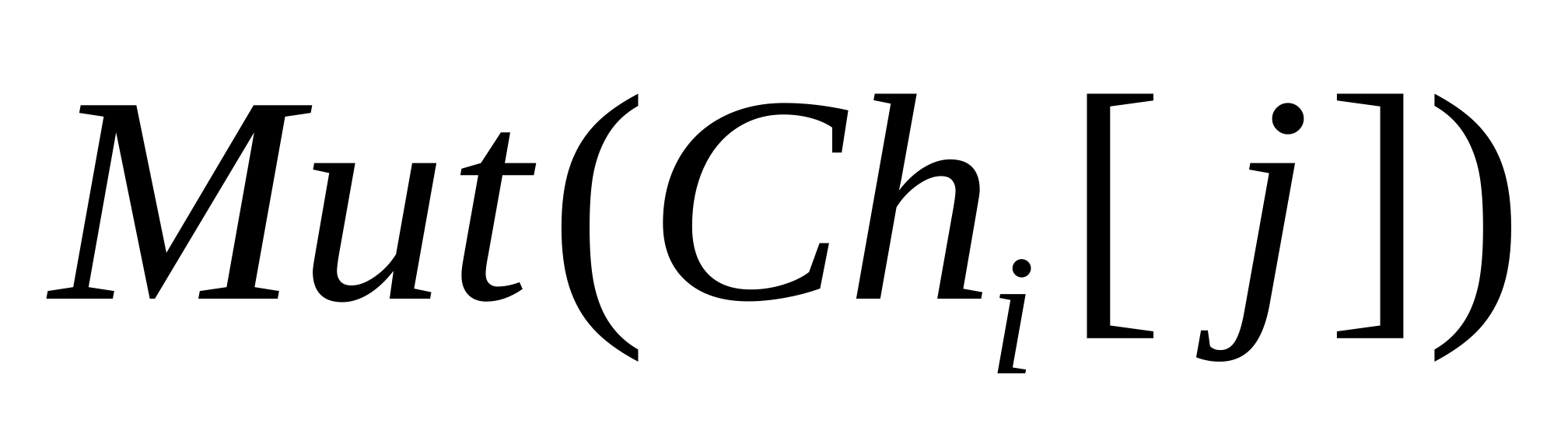
 - вектор особей (хромосом);

 - вектор приспособленности;

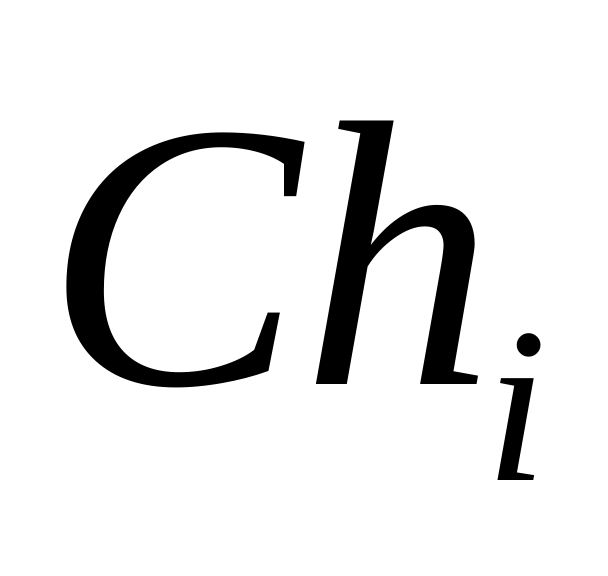
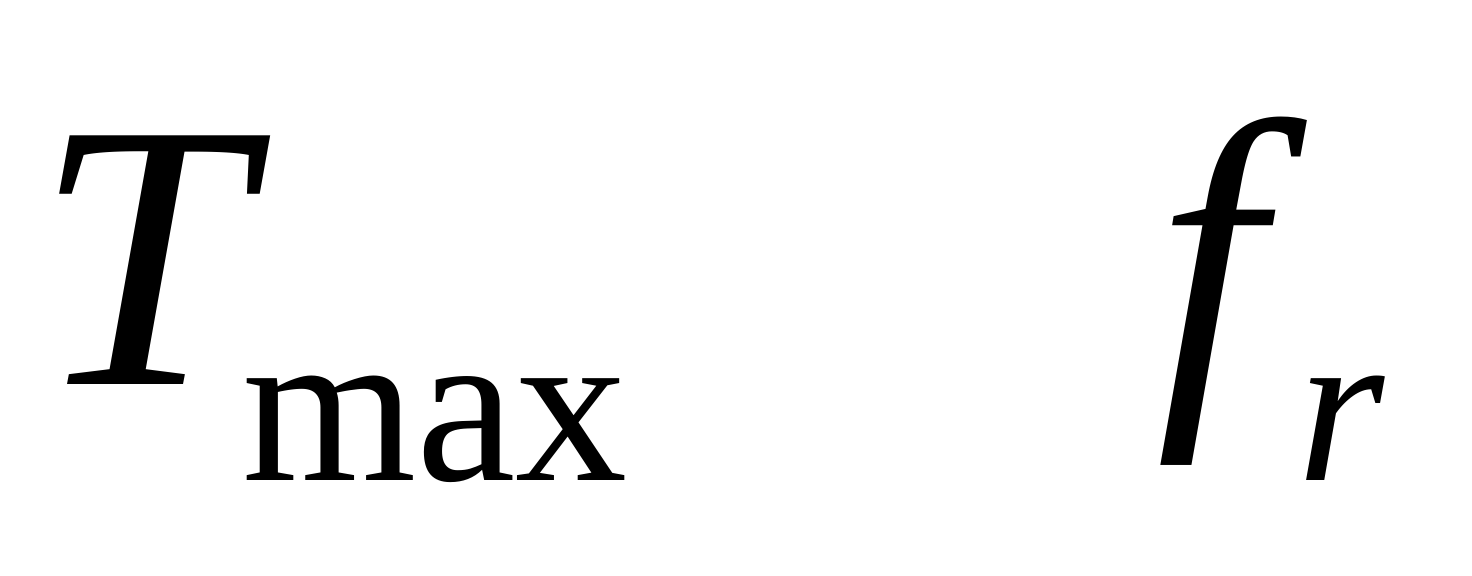
 - вероятность оператора кроссовера;

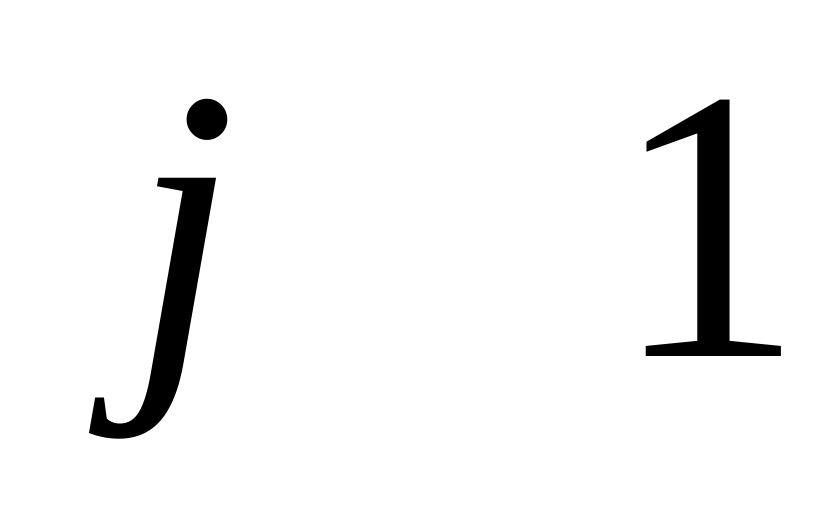
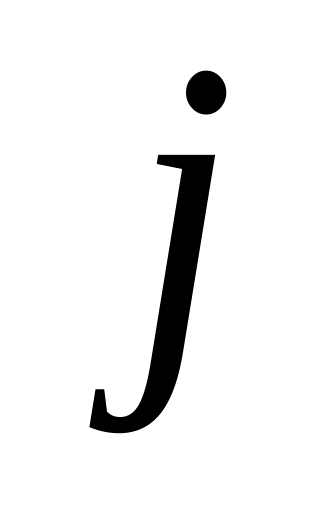
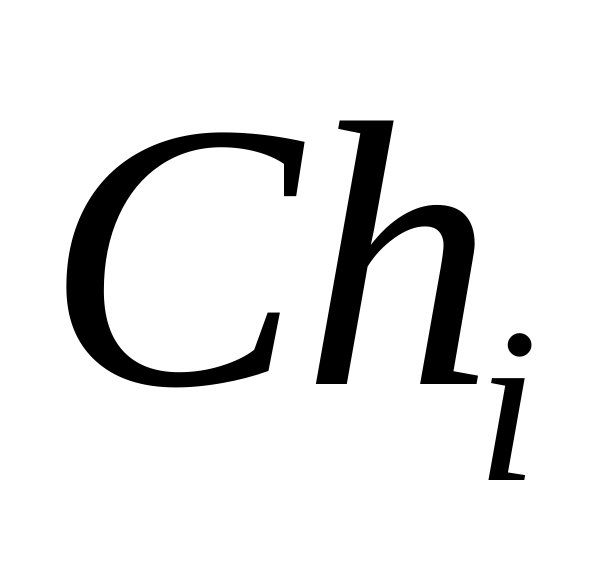
 - оператор кроссовера;

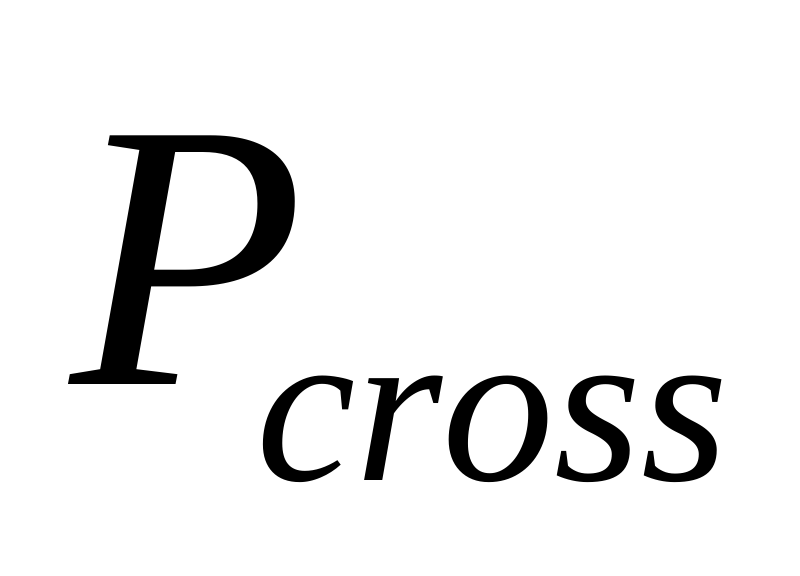
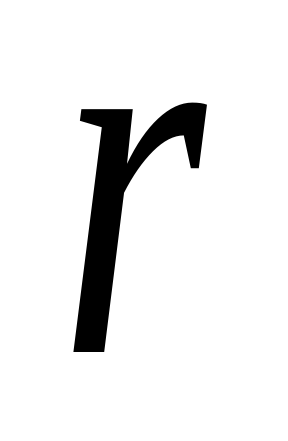
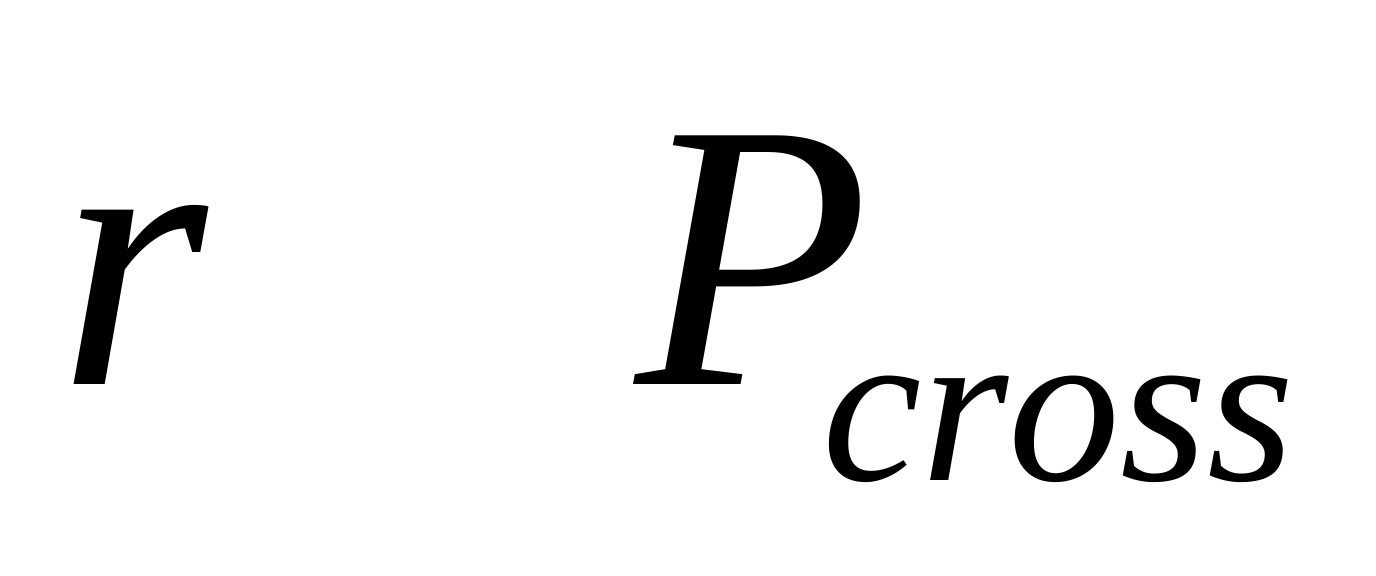
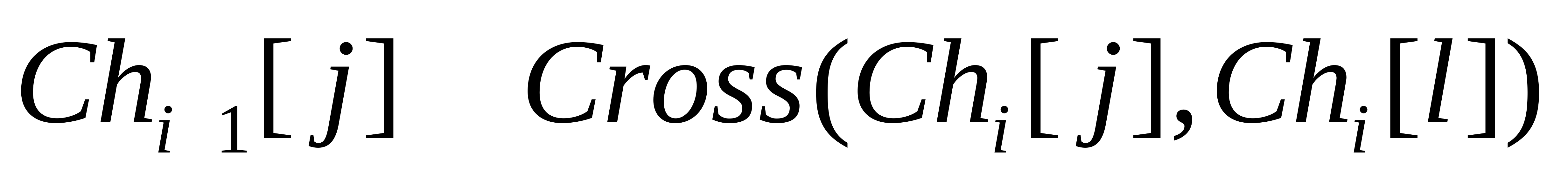
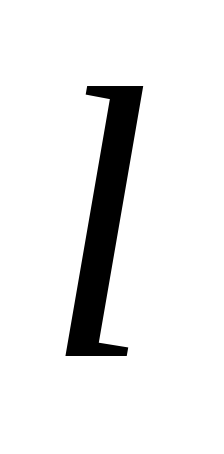
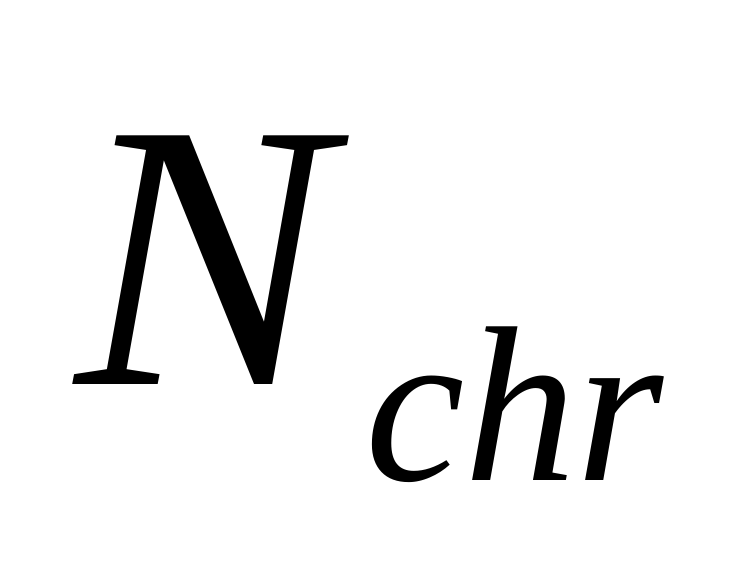
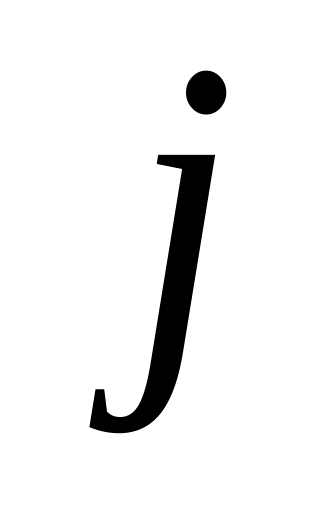
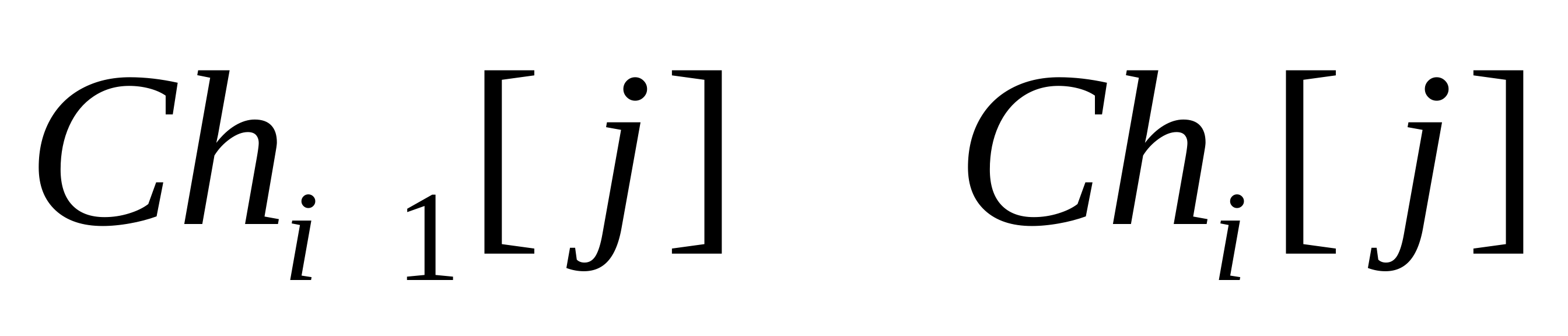
 - вероятность оператора мутации;

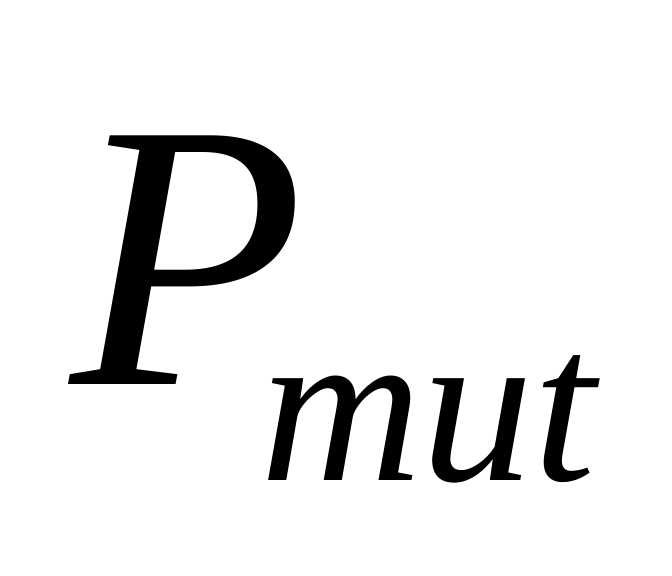
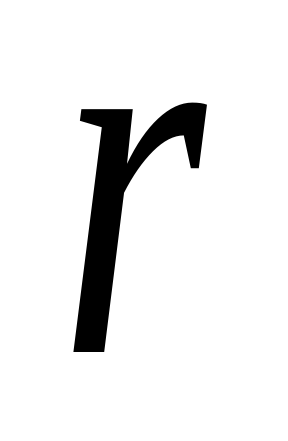
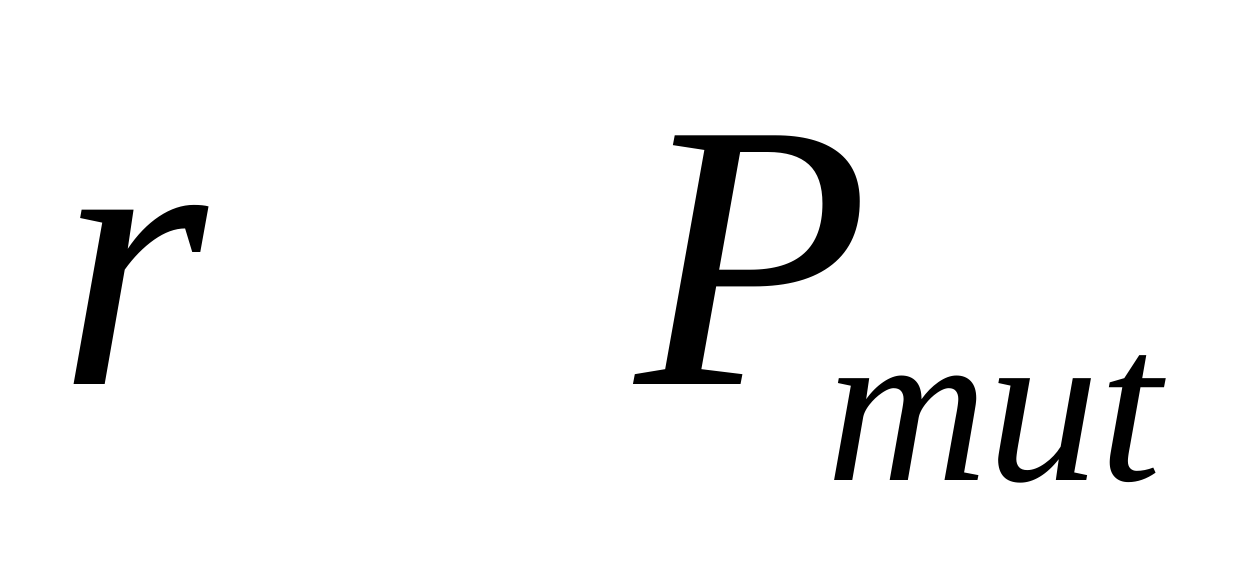
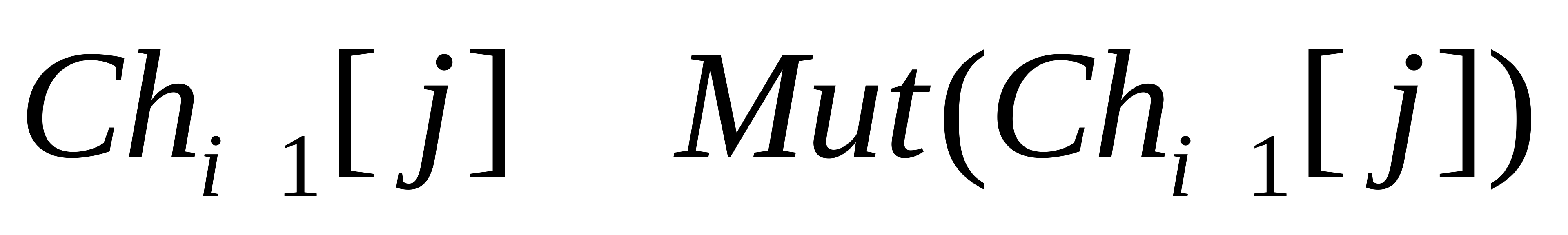
 - оператор мутации.

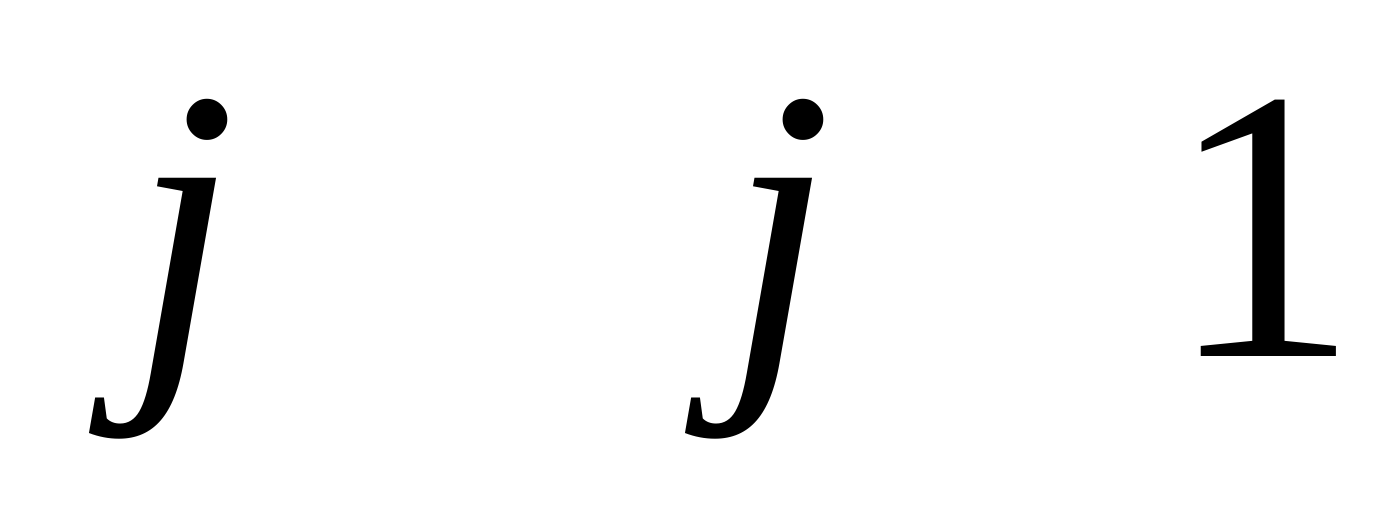
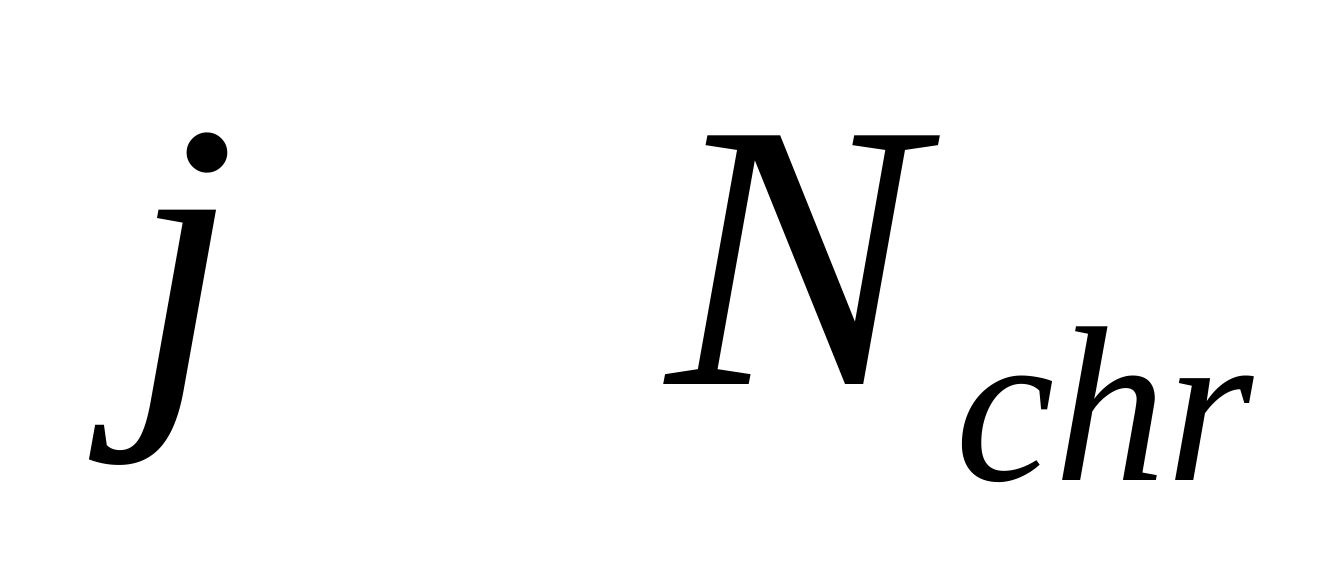
Рассмотрим подробнее Ш.2 алгоритма:

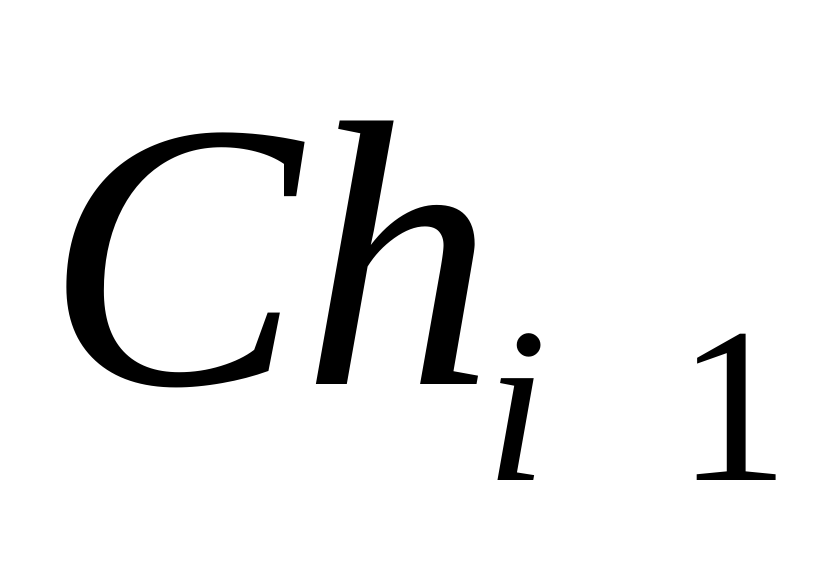
Ш.2.1 На вход шага поступает исходный вектор особей (хромосом) , каждая из особей представляет какое-то расписание с соответствующим ему  и соответственно приспособленностью особи.

Ш.2.2 , где  - порядковый номер особи в векторе .

Ш.2.3 Выполняется оператор кроссовера с заданной вероятностью . Для этого генерируется случайное число  в интервале [0..1]. Если , то , где - случайное число в интервале [1..], исключая совпадение с числом , иначе особь переходит в новое поколение без изменений, т.е. .

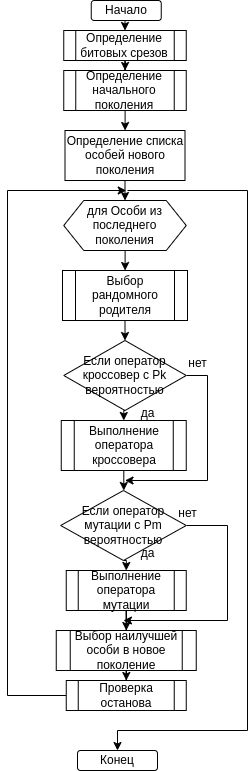
Ш.2.4 Выполняется оператор мутации с заданной вероятностью . Для этого генерируется случайное число  в интервале [0..1]. Если , то .

Ш.2.5 . Если , то перейти к Ш.2.3.

Ш.2.6 Сформировано новое поколение . Осуществляется переход на Ш.3.

Схематически работа данного алгоритма представлена в разделе <<Блок-схема>>

**Блок-схема**



Подробный листинг алгоритма приведен в приложении A.1

Пример результата работы программы изображен на рис 1.

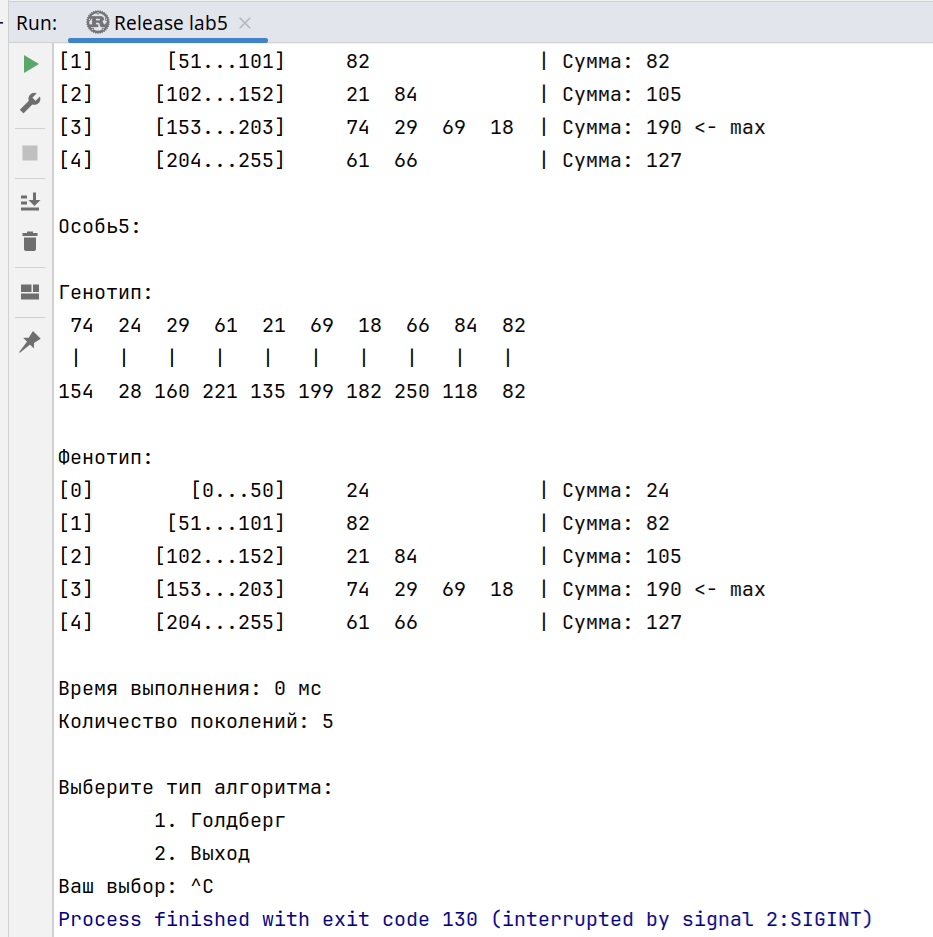


Рисунок 1 – Результат работы алгоритма

### **Вывод**

В рамках данной лабораторной работы было рассмотрено значимое направление исследований, связанных с решением однородной минимаксной задачи.

### **Список литературы**

1. Коффман Э.Г. Теория расписания и вычислительные машины. – M.: Наука, 1987.

### **Приложение A**

Листинг A.1 – Исходный код алгоритма

use std::cmp::min;

use rand::Rng;

use crate::operators::{crossover, mutation};

use crate::utils::Phenotype;

fn vec\_column<T: Copy>(matrix: &Vec<Vec<T>>, column: usize) -> Vec<T> {

matrix.iter().map(|row| row[column]).collect()

}

fn genes\_to\_string(genes: &Vec<[u8; 2]>) -> String {

let mut first\_row = String::new();

let mut second\_row = String::new();

let mut third\_row = String::new();

for gene in genes.iter() {

first\_row.push\_str(&format!("{:^4}", gene[0]));

second\_row.push\_str(&format!("{:^4}", "|"));

third\_row.push\_str(&format!("{:^4}", gene[1]));

}

format!("{}\n{}\n{}", first\_row, second\_row, third\_row)

}

pub fn main(matrix: &Vec<Vec<u32>>, k: u32, z: u32, p\_k: u32, p\_m: u32) {

let mut rnd = rand::thread\_rng();

let byte\_slices = {

let start: u8 = 0;

let end: u8 = 255;

let segments = matrix[0].len();

let step = ((end as usize - start as usize) / segments) as u8;

let mut byte\_slice: Vec<(u8, u8)> = vec![];

println!("\nСегменты байта:");

let mut segment\_start = 0;

let mut segment\_end = 0;

for (index, value) in (start..end).step\_by(step as usize).enumerate() {

segment\_start = min(value, end);

if index == segments - 1{

segment\_end = end;

} else {

segment\_end = min(value + step - 1, end);

}

byte\_slice.push((segment\_start, segment\_end));

println!("Сегмент: [{}] {} - {}", value, segment\_start, segment\_end);

}

byte\_slice

};

let mut generations: Vec<Vec<(Vec<[u8; 2]>, Phenotype)>> = vec![];

println!("\nНачальное поколение: ");

let start\_generation = {

let mut generation = vec![];

for i in 0..z {

let mut genotype: Vec<[u8; 2]> = vec![];

println!("\nОсобь{} Генотип: ", i + 1);

let column = vec\_column(matrix, 0);

for el in column.iter() {

genotype.push([\*el as u8, rnd.gen\_range(0..255)]);

}

println!("{}", genes\_to\_string(&genotype));

print!("\nФенотип: \n");

let phenotype = Phenotype::new(&byte\_slices, &genotype);

phenotype.print();

generation.push((

genotype,

phenotype

));

}

generation

};

generations.push(start\_generation);

let start\_time = std::time::Instant::now();

let mut gen\_counter = 0;

loop {

let last\_generation = generations.last().unwrap();

let mut new\_generation: Vec<(Vec<[u8; 2]>, Phenotype)> = vec![];

println!("\n------------- Формирование нового поколения №{} -------------", gen\_counter + 1);

for (g1\_index, genotype1) in last\_generation.iter().enumerate() {

let great\_child= {

let (genotype2, g2\_index) = {

let mut rnd\_genotype2 = rnd.gen\_range(0..last\_generation.len());

while rnd\_genotype2 == g1\_index {

rnd\_genotype2 = rnd.gen\_range(0..last\_generation.len());

}

(&last\_generation[rnd\_genotype2], rnd\_genotype2)

};

println!("\n> - - - - - - -Скрещивание особей {} и {}", g1\_index + 1, g2\_index + 1);

println!("\nОсобь{} Генотип: ", g1\_index + 1);

println!("{}", genes\_to\_string(&genotype1.0));

println!("Определитель фенотипа: {}", genotype1.1.max\_sum);

println!("\nОсобь{} Генотип: ", g2\_index + 1);

println!("{}", genes\_to\_string(&genotype2.0));

println!("Определитель фенотипа: {}", genotype2.1.max\_sum);

let mut child1 = genotype1.clone();

let mut child2 = genotype2.clone();

if rnd.gen\_range(0..100) < p\_k {

println!("Выполнился оператор кроссовера с вероятностью {}%", p\_k);

let (new\_genotype1, new\_genotype2) = crossover(&genotype1.0, &genotype2.0);

child1.0 = new\_genotype1;

child2.0 = new\_genotype2;

println!("\nОсобь{} Генотип: ", g1\_index + 1);

println!("{}", genes\_to\_string(&child1.0));

println!("Особь{} Генотип: ", g2\_index + 1);

println!("{}", genes\_to\_string(&child2.0));

}

if rnd.gen\_range(0..100) < p\_m {

println!("Выполнился оператор мутации с вероятностью {}%", p\_m);

child1.0 = mutation(&child1.0);

child2.0 = mutation(&child2.0);

println!("\nОсобь{} Генотип: ", g1\_index + 1);

println!("{}", genes\_to\_string(&child1.0));

println!("Особь{} Генотип: ", g2\_index + 1);

println!("{}", genes\_to\_string(&child2.0));

}

child1.1 = Phenotype::new(&byte\_slices, &child1.0);

child2.1 = Phenotype::new(&byte\_slices, &child2.0);

if child1.1.max\_sum < child2.1.max\_sum {

child1

} else {

child2

}

};

new\_generation.push(great\_child);

}

if generations.len() >= k as usize {

let mut last\_greet: Vec<u32> = vec![];

for index in (generations.len() - k as usize)..generations.len() {

let last\_gen = &generations[index];

let min\_max\_sum = last\_gen.iter().map(|el| el.1.max\_sum).min().unwrap();

last\_greet.push(min\_max\_sum);

}

println!(

"Последние {} поколений имеют лучший определитель фенотипа {:?} соответственно",

k,

last\_greet

);

if last\_greet.iter().all(|&x| x == last\_greet[0]) {

println!("Остановка алгоритма: последние {} поколений имеют одинаковый определитель фенотипа лучшей особи", k);

break;

}

}

generations.push(new\_generation);

gen\_counter += 1;

}

let delta\_time = start\_time.elapsed().as\_millis();

println!("\n--------------- Последнее поколение ---------------");

for (index, generation) in generations.last().unwrap().iter().enumerate() {

println!("\nОсобь{}: ", index + 1);

println!("\nГенотип: ");

println!("{}", genes\_to\_string(&generation.0));

print!("\nФенотип: \n");

generation.1.print();

};

println!("\nВремя выполнения: {:?} мс", delta\_time);

print!("Количество поколений: {}\n", gen\_counter + 1);

}