


# 1. Стандартные числовые функции на массивах и родственные им в языке Julia

## Числовые функции на массивах

Стандартная библиотека Julia содержит множество удобных функций для работы с массивами. Наиболее часто используемые сведены в следующей таблице.

Функция	Назначение	Пример
<code>sum(A)</code>	Сумма всех элементов массива	<code>sum([1, 2, 3]) → 6</code>
<code>prod(A)</code>	Произведение элементов	<code>prod([2, 3, 4]) → 24</code>
<code>cumsum(A)</code>	Накопленная сумма	<code>cumsum([1, 2, 3]) → [1, 3, 6]</code>
<code>cumprod(A)</code>	Накопленное произведение	<code>cumprod([2, 3, 4]) → [2, 6, 24]</code>
<code>maximum(A)</code>	Максимум всех элементов	<code>maximum([1, 7, 3]) → 7</code>
<code>minimum(A)</code>	Минимум всех элементов	<code>minimum([1, 7, 3]) → 1</code>
<code>max(x, y)</code>	Максимум из двух значений	<code>max(3, 5) → 5</code>
<code>min(x, y)</code>	Минимум из двух значений	<code>min(3, 5) → 3</code>
<code>findmax(A)</code>	(максимум, индекс)	<code>findmax([1, 7, 3]) → (7, 2)</code>
<code>findmin(A)</code>	(минимум, индекс)	<code>findmin([1, 7, 3]) → (1, 1)</code>
<code>argmax(A)</code>	Индекс максимума	<code>argmax([1, 7, 3]) → 2</code>
<code>argmin(A)</code>	Индекс минимума	<code>argmin([1, 7, 3]) → 1</code>

Следует отметить, что эти функции работают не только с массивами, но и с **произвольными итерируемыми объектами** (итераторами), реализующими интерфейс `iterate`.

 **Замечание 1.** Многие из этих функций, являются **функциями высшего порядка**, которые принимают в качестве первого аргумента некоторую функцию, применяемую ко всем элементам итерируемого аргумента.

Примеры:

```
sum(x->x^2, [1,2,3,4])
```

- вернет значение, равное  $1^2 + 2^2 + 3^2 + 4^2$ .

```
all(x -> x>0, [1, 2, 3])
true
```

- все элементы вектора положительные.

```
any(x-> x>0, [0, 2, 0])
true
```

- хотя бы один элемент вектора положительный.

📌 **Замечание 2.** Такие функции, как `sum`, `prod`, `maximum`, `minimum` и другие поддерживают работу не только с векторами, но и с **матрицами**. В каждой из них предусмотрен специальный ключевой параметр `dims`, через который можно явно указать, по какому измерению следует выполнять соответствующее действие. Например:

```
A = [1 2;
      3 4]
sum(A, dims=1) # - суммирует каждый столбец - результат строка
[4 6]

sum(A, dims=2) # - суммирует каждую строку - результат столбец
[3;
 7]
```

## 🔍 Функции поиска, фильтрации элементов подсчета их количеств

<code>findfirst(f, A)</code>	Первый индекс, где <code>f(x)</code> возвращает <code>true</code>	<code>findfirst(iseven, [1,3,4])</code> # 3
<code>findnext(f,A,i)</code>	Следующее вхождение после <code>i</code>	<code>findnext(iseven, [1,3,4,6], 2)</code> # 3
<code>findall(f,A)</code>	Индексы подходящих элементов	<code>findall(iseven, [1,2,3,4])</code> # [2, 4]

<code>filter(f,A)</code>	Элементы, для которых <code>f(x)</code> возвращает <code>true</code>	<code>filter(iseven, [1,2,3,4])</code> # [2,4]
<code>count(f, A)</code>	Количество подходящих элементов	<code>count(x -&gt; x&gt;2, [1,2,3,4])</code> # 2

## 2. Задача поиска индексов всех максимумов вектора (кортежа)

```
julia> vector = rand(8)
8-element Vector{Int64}:
 7
 8
 8
 7
 3
-6
-6
 0

julia> findall(x -> x==maximum(vector), vector)
2-element Vector{Int64}:
 2
 3
```

Данное решение с помощью стандартных функций имеет тот недостаток, что массив проходится дважды. Если необходимо в максимальной степени оптимизировать код, то потребуется написать свою собственную функцию:

```
function all_argmax(vector)
    maxval = typemin(eltype(vector))
    indices = Int[]
    for (i, x) in enumerate()
        if x > maxval
            maxval = x
            empty!(indices)
            push!(indices, i)
        elseif x == maxval
            push!(indices, i)
        end
    end
    return indices
end
```

## 3. Задача циклического сдвига элементов массива на заданное число позиций

Рассмотрим алгоритм циклического сдвига на 1 позицию вправо

```

function cycle_shift!(vector::AbstractVector)
    buf = vector[end]
    for i in reverse(firstindex(vector):lastindex(vector)-1)
        vector[i+1] = vector[i]
    end
    vector[begin] = buf
end

```

Повторяя этот алгоритм  $k$  раз, можно осуществить циклический сдвиг на  $k$  позиций. Пусть  $N$  - длина вектора, и пусть требуется осуществлять сдвиг на  $N/2$  позиций, тогда сложность такого алгоритма будет иметь асимптотическую оценку  $O(N^2)$

Однако такой сдвиг можно осуществить и за  $O(N)$  операций. Пусть по прежнему  $N$  - длина вектора, и необходимо осуществить сдвиг на  $k$  позиций. Идея алгоритма состоит в том, что надо разбить вектор на две соседние части `vector[1:k]` и `vector[k+1:N]`, к каждой из этих частей применить процедуру `reverse!`, а затем ту же процедуру `reverse!` применить ко всему массиву. Причем все это осуществлять на месте массива `vector` без дополнительных аллокаций. Если требуется осуществить сдвиг в противоположную сторону, то надо просто поменять последовательность указанных операций на противоположную.

```

function cycle_shift!(vector::AbstractVector, k::Integer)
    if k > 0
        reverse!(@view(vector[1:k]))
        reverse!(@view(vector[k+1:end]))
        reverse!(vector)
    elseif k < 0
        reverse!(vector)
        reverse!(@view(vector[1:-k]))
        reverse!(@view(vector[-k+1:end]))
    end
    vector
end

```

Имеется две стандартные функции `circshift!(vector,k)` и `circshift(vector,k)` осуществляющие циклический сдвиг массива на заданное число позиций в лево или в право - в зависимости от знака второго аргумента.

## 4. Стандартные функции сортировки

Сортировка элементов конечной последовательности (вектора) - это перестановка его элементов в порядке их не убывания (или не возрастания).

Таким образом, речь идет о сортировке массивов с элементами типа, для которого определена операция сравнения (функция `isless` - "меньше"). Не обязательно это только числовые типы.

Сортировки используются очень часто в различных алгоритмах. Например, если требуется найти, какой из элементов массива встречается наиболее часто, то для этого массив целесообразно вначале отсортировать. Другой пример - это алгоритм бинарного (быстрого) поиска. Если массив заранее отсортирован, то проверить, содержит ли массив некоторое значение можно всего за  $O(\log(N))$  операций, а не за  $O(N)$ , как было бы в случае неотсортированного массива. В самом деле, в случае отсортированного массива проверяемое значение надо сначала сравнить с серединой массива, если значение больше серединного, то дальнейший поиск надо осуществлять только в правой половине массива, иначе - в левой. Таким образом, размер области поиска на каждом шаге алгоритма уменьшается вдвое.

В стандартной библиотеке Julia имеются следующие функции сортировки: `sort!`, `sort`, `sortperm!`, `sortperm`. Первые две из которых возвращают отсортированный массив, а вторые две - перестановку индексов отсортированного массива, в этой перестановке индексов, первоначальные индексы следуют уже в новом порядке, соответствующем новым положениям элементов отсортированного вектора.

Функции, заканчивающиеся восклицательным знаком, сортируют массив на месте, функции без восклицательного знака в конце сортируют копии массивов, оставляя исходные массивы в неизменном виде.

Все эти функции реализуют самые эффективные алгоритмы сортировки, имеется также возможность выбора алгоритма сортировки, а также определить различные другие параметры (которых довольно много у каждой функции).

## Простейшие алгоритмы сортировки квадратичной сложности

### Пузырьковая сортировка

```
function bubble_sort!(series)
    n = lastindex(series)
    for i in Iterators.take(series, n-1) # можно 1:n-1 - для стандартных
массивов
        #УТВ: i-1 последних элементов стоят на "своих" местах
        for j in firstindex(series):n-i # 1:n-1 - для стандартных массивов
            if series[j] > series[j+1]
                series[j], series[j+1] = series[j+1], series[j]
            end
        end
    end
end
```

```

        end
    return series
end
end

```

Чтобы функция сортировки не изменяла массив, ее можно реализовать так:

```

bubble_sort(series) = bubble_sort!(copy(series))

```

**Замечание.** Если элементы массива представляют собой структуры, содержащие поля, ссылающиеся на другие объекты, то может показаться, в этом случае необходимо глубокое копирование. Однако глубокое копирование и в этом случае тоже не требуется.

Даже если требуется оставить исходный массив в неизменном виде, то должна быть создана его поверхностная копия, и эту копию надо сортировать. Объекты, на которые ссылаются поля элементов сортируемого вектора будут при этом всегда оставаться в памяти неподвижно на своих первоначальных местах. При поверхностном копировании перемещаться будут только ссылающиеся на них элементы сортируемого вектора (копии исходного вектора), и этого достаточно.

## Сортировка вставками

```

function insert_sort!(series)
    for i in Iterators.drop(series,1)
        #УТВ: i - firstindex(series) первых элементов стоят на "своих" местах
        j = i
        while j > firstindex(series) && series[j] > series[j-1]
            series[j], series[j-1] = series[j-1], series[j]
            j -= 1
        end
    return series
end
end

```

## Задача получения перестановок индексов вектора, совершаемых при сортировке его элементов

```

function insert_sortperm!(series)
    perm = collect(firstindex(series):lastindex(series))
    n = length(series)
    for i in Iterators.drop(series,1)
        #УТВ: i - firstindex(series) первых элементов стоят на "своих" местах

```

```

        j = i
        while j > firstindex(series) && series[j] > series[j-1]j
            seres[j], series[j-1] = series[j-1], series[j]
            perm[j], perm[j-1] = perm[j-1], perm[j]
            j -= 1
        end
        return perm
    end
end

```

## Сортировка не числовых последовательностей

Пример, отсортировать последовательность индексов столбцов матрицы в порядке не убывания суммы их элементов.

```

function sortperm_colon(vecfun::Function, A::AbstractMatrix)
    perm = collect(first(axes(A,2)):last(axes(A,2)))
    sort!(perm, by = j->vecfun(@view(A[:,j])))
    # ключевой параметр by позволяет задать функцию, определяемую на членах
    # сортируемой последовательности, и определяющую, по какому критерию
    # сравниваются члены этой последовательности
end

A = randn(10, 1000)
perm = sortperm_colon(sum, A)

```

## Алгоритм сортировки подсчетом линейной сложности

Этот способ сортировки применим не во всех случаях, например, его легко применить, когда элементы массива имеют целочисленный тип ограниченного размера (не BigInt). Идея сортировки состоит в том, что сначала определяются наименьшее и наибольшее значения в массиве. Поскольку эти значения целые, то их можно использовать в качестве границ индексов нового массива, который может быть создан и проинициализирован нулями (целого типа).

Таким образом, каждый элемент исходного массива потенциально рассматривается как значение индекса нового массива, предназначенного для подсчета числа вхождений каждого значения исходного массива.

А именно, исходный массив перебирается с первого элемента до последнего, и поскольку каждое значение этого массива есть некоторый индекс нового вспомогательного массива, то можно подсчитать, сколько раз каждый элемент исходного массива повторяется в

этом массиве. А затем просто создать новый массив, продублировав каждый элемент исходного массива нужное число раз и в порядке не убывания их значений.

```
using OffsetArrays

function count_sort!(vector::AbstractVector{Int})
    imin, imax = extrema(vector)
    counters = OffsetArray(zeros{Int, imax-imin+1}, imin:imax)
    for i in vector
        counters[vector[i]] += 1
    end
    i = 0
    for k in echindex(counters)
        n = counters[k]
        while n > 0
            n -= 1; i += 1
            vector[i] = k
        end
    end
    vector
end
```

## 5. Стандартный пакет LinearAlgebra

Операции и процедуры линейной алгебры в Julia реализованы в стандартном пакете (его не надо устанавливать, надо только импортировать) имеется специальный пакет `LinearAlgebra`. Не будем останавливаться на функциях этого пакета, с ним лучше знакомиться при изучении численных методов линейной алгебры.

В части рассмотрения вычислительных алгоритмов линейной алгебры мы ограничимся только разработкой алгоритма Жордана-Гаусса приведения матрицы к ступенчатому виду и алгоритмов, в основе которых лежит данный алгоритм.

## 6. Проектирование алгоритма Жордана-Гаусса

```
function transform_to_steps!(A::AbstractMatrix; epsilon = 1e-7)
    @inbounds for k ∈ 1:size(A, 1)
        absval, Δk = findmax(abs, @view(A[k:end, k]))
        (absval <= epsilon) && throw("Вырожденная матрица")
        Δk > 1 && swap!(@view(A[k, k:end]), @view(A[k+Δk-1, k:end]))
        for i ∈ k+1:size(A, 1)
            t = A[i, k] / A[k, k]
            @. @views A[i, k:end] = A[i, k:end] - t * A[k, k:end]
        end
    end
end
```



```
end  
return A  
end
```

```
@inline function swap!(A,B)  
    @inbounds for i in eachindex(A)  
        A[i], B[i] = B[i], A[i]  
    end  
end
```

**Замечание.** Значение ключевого параметра `epsilon` функции `transform_to_steps!` содержит пороговое значение для определения нуля (в приближенном смысле). При этом алгоритм записан так, что если в столбце не обнаруживается ни одного не нулевого элемента (что означает, что матрица вырожденная), то выбрасывается исключение. Однако, если приведение к ступенчатому виду осуществляется , например, с целью вычисления ранга матрицы, то прерывать процедуру не надо, ее следует довести до конца.