Алгоритмы сортировки (продолжение)

1. Алгоритмы улучшающие, простейшие алгоритмы сортировки квадратичной сложности

Идея повышения эффективности алгоритмов сортировки квадратической сложности (без изменения этой асимптотической оценки сложности) основывается на том, что операции сравнения являются далеко не самыми затратными. Более затратными являются операции копирования значений элементов массива (выполняемы при необходимости изменить их порядок следования). Поэтому, если каким-либо способом удастся сократить число таких копирований, то это может существенно повысить эффективность сортировки.

Так, в случае пузырьковой сортировки, чем больше в начале массива имеется больших по величине элементов, тем большее число копирований придется сделать при перемещении их в конец массива. Имеется модификации алгоритма пузырьковой сортировки, называема сортировкой расческой, которая за счет некоторого улучшения стратегии снижает количество необходимых копирований.

Аналогично дело обстоит и при сортировке вставками, улучшенной модификацией которого является алгоритм Шелла.

1.1. Алгоритм сортировки "расческой"

Идея это модификации пузырьковой сортировки состоит в том, чтобы осуществлять пузырьковую сортировку много кратно, применяя ее к подмассиву, получаемому из основного массива путем выборки из него элементов с некоторым постоянным шагом. И этот шаг постепенно уменьшается от предельно большого значения к 1. За счет этого многие большие значения как бы "вычесываются" заранее, и когда шаг выборки становится наконец равным 1, в начале массива больших значений оказывается уже совсем не много.

На каждой новой итерации шаг прореживания уменьшается в 1.247 раз - это эмпирически оптимизированное значение.

```
function comb_sort!(a::AbstractVector; factor=1.2473309)
    step = length(a)-1
    while step >= 1
        for i in firstindex(a):lastindex(a)-step
            if a[i] > a[i+step]
```

```
                a[i], a[i+step] = a[i+step], a[i]
            end
        end
        step = Int(floor(step/factor))
    end
    # Теперь массив почти упорядочен, осталось сделать всего
    # несколько итераций внешнего цикла в bubble_sort!(a)
    bubble_sort!(a)
end
```

Сортировка расческой является весьма эффективным и на реальных данных может конкурировать с быстрыми сортировками с оценкой сложности $O(n \log n)$, хотя его асимптотическая сложность оценивается всего лишь как $O(n^2)$.

Визуализацию работы алгоритма сортировки расческой можно посмотреть, например, здесь https://ru.wikipedia.org/wiki/Сортировка_расчёской

1.2. Алгоритм Шелла

Этот алгоритм есть модификация алгоритма вставками. Однако, в отличие от классических вставок, здесь последовательно сортируется прореженный с шагом `step` массив. Величина этого шага начинается с половины длины исходного массива (примерно) до 1, на каждой новой итерации уменьшаясь вдвое (примерно).

Визуализацию процесса сортировки Шелла можно посмотреть, например, здесь https://ru.wikipedia.org/wiki/Сортировка_Шелла

Величину шага прореживания от итерации к итерации можно вычислять, например, так

```
julia> n=1000;
julia> [n÷2^i for i in 1:Int(floor(log2(n)))]

10-element Vector{Int64}:
  500
  250
  125
   62
   31
   15
    7
    3
    1
```

Соответственно, алгоритм Шелла может быть записан следующим образом

```
function shell_sort!(a::AbstractVector)
    n=length(a)
    step_series = (n÷2^i for i in 1:Int(floor(log2(n))))
    # при n=1000: 500, 250, 125, 62,31, 15, 7, 3, 1
    for step in step_series
        for i in firstindex(a):lastindex(a)-step
        # - тут записано без привязки к способу индексации массива
        #   (имеется ввиду индексация с 0, или с 1, или ещё как)
            j = i
            while j >= firstindex(a) && a[j] > a[j+step]
                a[j], a[j+step] = a[j+step], a[j]
                j -= step
            end
        end
    end
    return a
end
```

Здесь последовательность шагов прореживания массива задается не заранее сформированным массивом, а генератором
`step_series = (n÷2^i for i in 1:Int(floor(log2(n))))`
который демонстрирует "ленивое" поведение, вычисляя очередное значение по мере надобности.

Замечание. Ту же идею можно было бы реализовать, и непосредственно используя функцию `insert_sort!` в качестве подпрограммы:

```
function shell_sort_alt!(a::AbstractVector)
    n = length(a)
    step_series = (n÷2^i for i in 1:Int(floor(log2(n))))
    for step in step_series
        for i in 1:step-1
            insert_sort!(@view(a[begin-1+i:step:end])) # - сортировка
вставками выделенного (прореженного) подмассива
        end
    end
    return a
end
```

Однако этот вариант сортировки будет выполняться значительно медленнее предыдущего. Объясняется это тем, что здесь используются представления не плотных срезов (получающихся при `step > 1`) , которые не эффективны.

2. Алгоритмы сортировки вычислительной сложности $N \log N$

Начнем с так называемого алгоритма сортировки слияниями. Его основная идея состоит в том, что исходный массив надо каким-либо способом многократно делить на части, каждую часть сортировать отдельно, а затем отсортированные части объединять в один отсортированный массив.

2.1. Базовая вспомогательная процедура сортировки слияниями

```
"""
merge!(a1, a2, a3)::Nothing
    ДАНО:
    length(a3) == length(a1)+length(a2) && issorted(a1) && ssorted(a2)

    РЕЗУЛЬТАТ: issorted(a3)
"""
@inline
function Base.merge!(
    a1::AbstractVector{T},
    a2::AbstractVector{T},
    a3::AbstractVector{T}
)::Nothing where T
    i1, i2, i3 = firstindex(a1), firstindex(a2), firstindex(a3)
    @inbounds # - передотвращает проверки выхода за пределы массивов
    while i1 <= lastindex(a1) && i2 <= lastindex(a2)
        if a1[i1] < a2[i2]
            a3[i3] = a1[i1]
            i1 += 1
        else
            a3[i3] = a2[i2]
            i2 += 1
        end
        i3 += 1
    end
    @inbounds
    if i1 > lastindex(a1)
        a3[i3:end] .= @view(a2[i2:end])
        # !! тут перед = точка должна должна быть внизу
        # Если бы было без точки, то это привело бы к лишним
        # аллокациям (к созданию промежуточного массива)
    else
        a3[i3:end] .= @view(a1[i1:end])
```

```
    end
    nothing
end
```

Макрос `@inline` - делает функцию "встраиваемой", т.е. во время компиляции ее тело будет встроено непосредственно в код вызывающей функции (за счет этого происходит экономия на времени, затрачиваемым на вызов функции; это время очень небольшое, но тем не менее)

2.2. Алгоритм сортировки слияниями

Этот алгоритм использует два массива, копируя из одного в другой, и каждый раз путем слияния вдвое увеличивая длину уже отсортированных на предыдущем шаге блоков (грубо говоря), пока в каком-то из двух массивов не останется только один отсортированный блок (совпадающий с этим массивом). Первоначально массив делится на блоки по одному элементу в каждом (такие блоки можно считать отсортированными).

```
function merge_sort!(a)
    b = similar(a) # - вспомогательный массив того же размера и типа, что и
массив a
    N = length(a)
    n = 1 # n - текущая длина блоков
    @inbounds
    while n < N
        K = div(N,2n) # - число имеющихся пар блоков длины n
        for k in 0:K-1
            merge!(@view(a[(1:n).+k*2n]),
            @view(a[(n+1:2n).+k*2n]), @view(b[(1:2n).+k*2n]))
        end
        if N - K*2n > n
        # - осталось еще смержить блок длины n и более короткий
        # остаток
            merge!(@view(a[(1:n).+K*2n]),
            @view(a[K*2n+n+1:end]), @view(b[K*2n+1:end]))
        elseif 0 < N - K*2n <= n
        # - оставшуюся короткую часть мержить уже не с чем
            b[K*2n+1:end] .= @view(a[K*2n+1:end])
        end
        a, b = b, a
        n *= 2
    end
    # - если цикл был выполнен нечетное число раз, то b - это
    # исходная ссылка на массив (на внешний массив), и a - это ссылка
    # на вспомогательный массив (локальный)
```

```
    if isodd(log2(n))
        b .= a # тут перед = точка должна быть внизу
        a = b
    end
    return a
end
```

3. Пирамидальная сортировка

Прежде, чем мы сможем перейти собственно к алгоритму пирамидальной сортировки, понадобится обсудить тип данных, называемый "кучей" и некоторые вязанные с ним алгоритмы.

3.1. Тип данных "куча"

В программировании исключительно важную роль играют таки структуры данных, как очередь, стек, очередь с приоритетами, различные деревья и т.п. Для языка Julia имеется специальный пакет `DataStrctures.jl`, где реализованы эти и многие другие структуры данных.

Сейчас мы кратко коснемся структуры данных, называемой **кучей**. Эта структура данных нас будет интересовать сейчас только с точки зрения возможности ее использования для сортировки массива.

Куча представляет собой двоичную иерархическую структуру (двоичное дерево), в которой значение каждого из двух дочерних элементов меньше значения родительского элемента. Это если речь идет о так называемой максимальной куче. Если же речь идет о так называемой минимальной куче, то значение каждого из двух дочерних элементов должно быть всегда больше значения родительского элемента.

Таким образом, если на вершине кучи (в корне дерева) находится максимальное значение, то такая куча максимальная. В противном случае куча - минимальная.

Не надо путать понятие кучи с упорядоченным двоичным деревом, в котором для любой вершины верно, что все элементы, ассоциированные с левым поддеревом, меньше элемента ассоциированного с данной вершиной, а все элементы, ассоциированные с правым поддеревом больше этого элемента.

3.2. Реализация кучи на базе массива

Как правило кучу реализуют на базе обычного массива. При этом считается, что каждый i -й элемент массива имеет (если имеет) два дочерних элемента с индексами $2i$ и $2i+1$.

Тогда говорят, что массив `heap` имеет структуру **кучи** (максимальной), если для каждого `ero i`-го элемента верно, что:

- `heap[i] >= heap[2i]`
- `heap[i] >= heap[2i+1]`

разумеется, что рассматривается индекс `i` превосходящий величины `length(heap)÷2`.

Преобразовать произвольный массива в максимальную кучу можно за $O(N)$ действий следующим образом

```
function heap!(array)
    N = length(array)
    for i in 1:N÷2
        if array[i] < array[2i]
            array[i], array[2i] = array[2i], array[i]
        end
        if 2i+1 <= N && array[i] < array[2i+1]
            array[i], array[2i+1] = array[2i+1], array[i]
        end
    end
    return array
end
```

Соответственно, если при всех допустимых `i` выполняются условия
- `heap[i] <= heap[2i]`
- `heap[i] <= heap[2i+1]`
то массив `heap` имеет структуру **минимальной** кучи.

3.3. Перемещение элемента на место, соответствующее его приоритету

Если сформирована, минимальная куча, то на вершине кучи (в первой позиции массива) находится наименьший элемент массива. В этом смысле можно считать, что он имеет наивысший приоритет, если по какой-то причине важно иметь возможность всегда извлекать из очереди элемент с наименьшим значением (так устроенная очередь называется очередью с приоритетом).

Допустим, что имеется только один единственный элемент в куче, который стоит на позиции (в общем случае - `i`-ой), не соответствующей его приоритету.

Такая ситуация может возникать, например, при добавлении/удалении элемента в кучу. А также, если при решении какой-либо задачи, на каком-то шаге произошло изменение

значения приоритета какого-то одного элемента кучи.

Задача состоит в том, что бы переместить этот элемент на нужную позицию, восстановив требуемую структуру кучи.

Например, может потребоваться "понизить" элемент с индексом `index`, перемещая его ближе к концу кучи до тех пор, пока он не займет свое "правильное" место в массиве ("куче"), соответствующее его значению.

Это можно сделать с помощью следующей процедуры, сложность которой оценивается, как $O(log(N))$.

```
"""
    down_first!(heap::AbstractVector)::Nothing

Перемещает первый элемент массива ближе к концу кучи, пока он не займет свое
"правильное" место в куче.
"""
function down_first!(heap::AbstractVector)::Nothing
    index = 1
    N = length(heap)
    while index < N÷2
        if heap[index] < heap[2index]
            heap[index], heap[2index] = heap[2index], heap[index]
        end
        if 2index+1 <= N && heap[index] < heap[2index+1]
            # порядок аргуметов лоической связки && здесь важен!
            heap[index], heap[2index+1] = heap[2index+1], heap[index]
        end
        index *= 2
    end
end
```

3.4. Алгоритм пирамидальной сортировки

Пусть $heap$ - произвольный вектор длины N.
Процедура сортировки состоит из следующих шагов.

1. Преобразуем исходный массив в кучу (с помощью функции heap!), на что потребуется $O(N)$ операций.
2. Поменяем местами первый элемент массива (после предыдущего шага он будет максимальным) с последним (тем самым последний элемент массива окажется на "своем" окончательном месте), и переместим новый первый элемент массива на

"правильное" место в куче. С использованием функции `down!` на это потребуется $O(\log N)$ элементарных операций.
3. Длину сортируемой части массива уменьшим на 1 и перейдем к пункту 2, и т.д. пока длина сортируемой части станет раной 3 (первые три элемента кучи всегда отсортированы).

Таким образом, сложность пирамидальной сортировки имеет оценку $O(N \log N)$.

```
function heap_sort!(heap::AbstractVector)
    heap = heap!(heap)
    N = length(heap)
    # ИНВАРИАНТ: heap[1:N] - это максимальная куча && heap[N+1:end] -
    #            это отсортированная часть массива
    while N > 3
        heap[1], heap[N] = heap[N], heap[1]
        N -= 1
        down_first!(@view(heap[1:N]))
    end
    return heap
end
```

4. Быстрая сортировка Хоара с асимптотической оценкой сложности в среднем - $O(N \log(N))$

Рассмотрим теперь алгоритм сортировки Хоара. На практике этот алгоритм считается одним из самых эффективных, поэтому именно за ним закрепилось название "алгоритмом быстрой сортировки".

Однако для этого алгоритма оценка сложности $O(N \log(N))$ справедлива только в средне-статистическом смысле. В худшем случае этот алгоритм может выродиться в алгоритм квадратической сложности.

4.1. Базовая процедура алгоритма Хоара

Пусть имеется массив A и значение одного из его элементов, равное b. Требуется переставить элементы в массиве A так, чтобы в нем сначала следовали все элементы, меньшие b, затем - все, равные b, а затем - все, большие b.

Причем алгоритмическая сложность этой процедуры должна оцениваться как $O(N)$, где `N=length(A)`, и дополнительные массивы использоваться не должны.

Решим эту задачу с использованием метода инварианта цикла. Напомним, что инвариантом цикла (с предусловием) называют утверждение (предикат), зависящее от

фазовых переменных цикла (т.е. переменных, которые могут изменяться в теле цикла), имеющее значение "истина" как до начала цикла, так и после любого числа его повторений.

Идея состоит в том, что для решения задачи сначала надо сформулировать подходящий инвариант цикла, а затем уже на этой основе сконструировать соответствующий цикл.

Пусть величины K, M, L (индексы массива A) такие, что

- $\forall i \in 1 : K \quad A[i] < b$
- $\forall i \in K+1 : L \quad A[i] == b$
- $\forall i \in M+1 : N \quad A[i] > b$

Эти три условия, выполняемые одновременно, и будут составлять инвариант цикла.

В инварианте цикла ничего не утверждается про элементы массива для диапазона индексов $L+1 : M$, этот диапазон индексов соответствует еще не "обработанной" части массива. В самом начале процедуры весь массив совпадает с этой "не обработанной" частью, а в конце - "необработанная" часть должна стать пустой.

```
function part_sort!(A, b)
    N = length(A)
    K=0; L=0; M=N
    #ИНВАРИАНТ: A[1:K] < b && A[K+1:L] == b && A[M+1:N] > b
    while L < M
        if A[L+1] == b
            L += 1
        elseif A[L+1] > b
            A[L+1], A[M] = A[M], A[L+1]
            M -= 1
        else # if A[L+1] < b
            L += 1; K += 1
            A[L], A[K] = A[K], A[L]
        end
    end
    return K, M+1
end
```

4.2. Рекурсивная процедура сортировки Хоара

```
function quick_sort!(A)
    if isempty(A)
        return A
    end
```

```
    N = length(A)
    K, M = part_sort!(A, A[rand(1:N)])
    # - "базовый" элемент массива выбирается случайнам образом
    quick_sort!(@view(A[1:K]))
    quick_sort!(@view(A[M:N]))
    return A
end
```

Оценить сложность алгоритма быстрой сортировки `quick_sort!` можно следующим образом. Оценка основывается на предположении о том, что **в среднем** размеры его частей `A[1:K]` и `A[M+1:N]`, подлежащих дальнейшей сортировке, но уже по отдельности, будут получаться приблизительно равными (если взять, например, случайным образом сформированный массив).

Для упрощения анализа условимся считать их равными $N/2$. Тогда всю процедуру сортировки можно будет представить двоичным деревом высоты $log_2(N)$ (каждый узел этого дерева фиксирует факт разделения рекурсивно сортируемой части массива на два подмассива равной длины). При этом корню этого дерева соответствует N операций сравнения, и каждому его последующему ярусу тоже соответствуют те же N операций сравнения.

Таким образом, потребуется всего $N \log_2 N$ операций, т.е. оценка сложности алгоритма сортировки в среднестатистическом смысле может быть выражена как $O(N \log(N))$.

5. Порядковые статистики, вычисление медианы массива

Пусть имеется числовой массив A. Его k-ой **порядковой статистикой** называется значение k-го элемента этого массива, которое получилось бы после реализации процедуры сортировки массива A.

Однако для вычисления k-ой порядковой статистики (k считается фиксированным) вовсе не обязательно сортировать массив. Существует алгоритм вычисления этой величины, имеющий сложность всего $O(N)$, т.е. быстрый алгоритм.

В самом деле, если индекс k задан, то требуемую процедуру вычисления k-ой порядковой статистики легко получить из рассмотренного выше алгоритма Хоара быстрой сортировки. В самом деле, вычисление k-ой порядковой статистики от процедуры сортировки Хоара будет отличаться лишь тем, что для последующей после частичной сортировки массива дальнейшая обработка массива должна будет производиться лишь только над одной из двух полученных его частей, а именно, той из них, диапазон индексов которой включает заданное k.

В результате, первая частичная сортировка даст $O(N)$ операций сравнения, вторая - $O(N/2)$, третья - $O(N/4)$ и т.д., что в сумме составит всего $O(N)$ операций.

Замечание. Минимальное и максимальное значения массива являются его 1-ой и N-ой порядковыми статистиками, соответственно. Однако, хотя сложность их вычисления также оценивается как $O(N)$, их вычисление обычным способом потребует вдвое меньше сравнений (не говоря уже о том, что оно вовсе не потребует перестановок элементов массива, что существенно более затратно по сравнению с просто операцией сравнения).

```
function order_statistics!(A::AbstractVector{T},i::Integer)::T where T
    function part_sort!(indexes_range::AbstractUnitRange, b)
        K = indexes_range[begin]-1 # 0 - при первом вызове
        L = indexes_range[begin]-1 # 0 - при первом вызове
        M = indexes_range[end]     # N - при первом вызове
        # ИНВАРИАНТ: A[indexes_range[begin]:K] < b &&
        #            A[K+1:L] == b && A[M+1:indexes_range[end]] > b
        while L < M
            if A[L+1] == b
                L += 1
            elseif A[L+1] > b
                A[L+1], A[M] = A[M], A[L+1]
                M -= 1
            else # if A[L+1] < b
                L += 1; K += 1
                A[L], A[K] = A[K], A[L]
            end
        end
        return indexes_range[begin]:K, M+1:indexes_range[end]
    end

    function find(indexes_range)
        left_range, right_range =
            part_sort!(indexes_range, A[rand(indexes_range)])
        # - "базовый" элемент массива выбирается случайно
        if i in left_range
            return find(left_range)
        elseif i in right_range
            return find(right_range)
        else
            return A[i]
        end
    end
    find(firstindex(A):lastindex(A))
end
```

```
order_statistics(A, i) = order_statistics!(copy(A), i)
```

В этой процедуре, в отличие от сортировки Хоара, при вычислении `i`-ой порядковой статистики необходимо знать только диапазон индексов той части массива, в которой следует искать эту порядковую статистику. Поэтому используемая здесь вспомогательная функция модифицирована таким образом, чтобы та часть исходного массива, которая должна подвергнуться процедуре частичной сортировки, определялась бы не ссылкой на срез, а именно диапазоном соответствующих индексов.

Медиана массива

Если длина N массива нечетная, то его медианой называется порядковая статистика с индексом $(N-1)/2$. В случае же четной длины массива его медианой можно считать среднее арифметическое двух порядковых статистик с индексами $N/2-1$ и $N/2+1$.

Поэтому алгоритм вычисления медианы массива также может основываться на быстром алгоритме вычисления порядковых статистик.

Медиана массива, наряду со средним (средним арифметическим) значением массива является важной статистической характеристикой содержащихся в нем данных. Например, медианная оценка более адекватно показывает уровень доходов, нежели среднее арифметическое.

6. Общие теоретические сведения о алгоритмах сортировки

Утверждение. Никакой алгоритм сортировки, гарантированно сортирующий любой массив,
, основанный только на многократном применении операции сравнения к парам элементов массива, не может иметь асимптотическую сложность, меньшую, чем `O(n log n)`.

Доказательство.

Для оценки алгоритмической сложности алгоритма сортировки, основанного на сравнениях, достаточно получить оценку для минимально необходимого числа операций сравнения. Пусть имеется некоторый алгоритм сортировки, который требуется не более `d` операций сравнения ("<") . Поскольку операция сравнения имеет один из двух возможных исходов (true/false), и этих d операций должно быть достаточно, чтобы различить любые две перестановки элементов массива из всех возможных n! (мы предполагаем наихудший вариант, когда все элементы массива разные), то должно

выполняться неравенство $2^d \geq n!$. В противном случае результат сортировки оказался бы не определенным - не более чем за d операций сравнения должна быть определена **ровно одна** перестановка из всех $n!$.

Таким образом, для числа операций сравнения имеем оценку снизу: $d \geq \log n!$
Далее, для оценки факториала воспользуемся асимптотической формулой Стирлинга

$$n! = \left(\frac{n}{e}\right)^n \sqrt{2\pi n}(1 + O\left(\frac{1}{n}\right), \quad n \to +\infty$$

В результате получим

$$d \geq n \log n - n \log e + \frac{1}{2}\log(2\pi n) = O(n \log n)$$

■

Замечание 1. Из данного утверждения не следует, конечно, принципиальная невозможность существования более быстрых алгоритмов сортировки. Более того, такие алгоритмы возможны, но если только они не основываются на сравнениях элементов - таковым, например, является алгоритм сортировки подсчетами.

Замечание 2. Алгоритмы имеющие одну и ту же асимптотическую оценку вычислительной сложности, могут различаться по практической эффективности. Это связано с тем, что нотация О-большое предполагает наличие коэффициентов в оценивающих выражениях, и эти коэффициенты для разных алгоритмов могут различаться весьма сильно. По этой причине может оказаться даже, что какой-то алгоритм с лучшей асимптотикой на практике будет менее эффективен в сравнении с некоторым алгоритмом с худшей асимптотикой.

Кроме вычислительной эффективности, характеризующей скорость выполнения, алгоритмы сортировки различаются также по требованиям к дополнительной памяти.

Еще одной важной характеристикой алгоритма сортировки, является его свойство сохранять или не сохранять порядок следования элементов массива с равными значениями.

Свойство алгоритма сохранять первоначальный порядок следования равных элементов называется **стабильностью**. Стабильность сортировки важна, например, в случаях, когда требуется отсортировать массив сначала по одному критерию, а потом по другому.

Например, пусть имеется список спортсменов, который требуется отсортировать сначала по росту спортсменов, а потом по их весу, так чтобы спортсмены следовали в порядке не убывания их роста, а спортсмены с равным ростом перечислялись бы в порядке не убывания их веса. Понятно, что для решения такой задачи годятся только стабильные сортировки.

Следующая таблица дает общее представление о наиболее популярных алгоритмах сортировки, включая самые современные, такие как radix sort.

Алгоритм	Сложность (время)	Память (доп.)	Стабильность	In-place	Комментар
Insertion Sort *(вставками)*	`O(n²)`, `O(n)` на почти отсортированных	`O(1)`	✅ Да	✅ Да	Очень эффективе маленьких массивах и почти отсортирова
Selection Sort *(выбором)*	`O(n²)`	`O(1)`	❌ Нет	✅ Да	Простой, но всегда медленный
Bubble Sort *(пузырьком)*	`O(n²)`	`O(1)`	✅ Да	✅ Да	Учебный, на практике не используетс
Merge Sort *(слиянием)*	`O(n log n)`	`O(n)`	✅ Да	❌ Нет	Стабильный хорош для списков и больших массивов
Quick Sort *(быстрая)*	`O(n log n)` avg, `O(n²)` worst	`O(log n)` (стек)	❌ Нет	✅ Да	Очень быст но может "сломаться" защиты
Heap Sort *(пирамидальная)*	`O(n log n)`	`O(1)`	❌ Нет	✅ Да	Используетс приоритетн очередях
Counting Sort *(подсчётом)*	`O(n + k)`	`O(k)`	✅ Да	❌ Нет	Почти лине если диапаз значений ма
Radix Sort *(разрядная)*	`O(n * d)`	`O(n + r)`	✅ Да (если с `counting sort`)	❌ / ✅	Быстрая дл чисел и фиксирован ключей
Bucket Sort *(по корзинам)*	`O(n + k)` avg	`O(n + k)`	❌ Часто нет	❌ Нет	Хороша при равномерно распределё значениях

Алгоритм	Сложность (время)	Память (доп.)	Стабильность	In-place	Комментар
TimSort *(адаптивная слиянием)*	`O(n log n)` worst, `O(n)` best	`O(n)`	✅ Да	❌ Нет	Использует Python и Jav эффективна реальных данных
IntroSort *(интроспективная)*	`O(n log n)` гарантированно	`O(log n)`	❌ Нет	✅ Да	Умный Quic с защитой с худших слу
Shell Sort *(Шелла)*	`O(n log² n)` ~ `O(n^1.5)`	`O(1)`	❌ Нет	✅ Да	Быстрее вставками, нестабильн непредсказ

📘 Краткие описания алгоритмов сортировки

- **Insertion Sort** *(вставками)*
  Простой и стабильный алгоритм, эффективно работает на небольших массивах или почти отсортированных данных. Вставляет каждый элемент в уже отсортированную часть.
- **Selection Sort** *(выбором)*
  Последовательно находит минимальный элемент и перемещает его в начало. Прост в реализации, но медленный и нестабильный.
- **Bubble Sort** *(пузырьком)*
  Обменивает соседние элементы, если они идут в неправильном порядке. Повторяет, пока массив не отсортирован. Учебный алгоритм, редко используется на практике.
- **Merge Sort** *(слиянием)*
  Разделяет массив на две половины, рекурсивно сортирует и сливает обратно. Стабильный, требует дополнительную память.
- **Quick Sort** *(быстрая)*
  Разделяет массив на части по опорному элементу. Очень быстрый на практике, но нестабильный и имеет худший случай `O(n²)` без защиты.
- **Heap Sort** *(пирамидальная)*
  Основан на структуре данных "куча". Всегда `O(n log n)`, работает на месте, но нестабильный и чуть медленнее QuickSort.
- **Counting Sort** *(подсчётом)*
  Подсчитывает количество вхождений каждого значения. Быстрая и стабильная при небольшом диапазоне значений. Не сравнительная.

- **Radix Sort** *(разрядная)*
  Сортирует поразрядно (обычно по байтам), применяя stable counting sort на каждом разряде. Очень быстрая на числах и фиксированной длине ключей.
- **Bucket Sort** *(по корзинам)*
  Делит значения на диапазоны (корзины), сортирует каждую корзину отдельно и сливает. Эффективна на равномерно распределённых данных.
- **TimSort** *(адаптивная слиянием)*
  Гибрид MergeSort и InsertionSort. Анализирует структуру массива и использует уже отсортированные участки. Очень эффективен на реальных данных. Используется в Python, Java.
- **IntroSort** *(интроспективная)*
  Начинает как QuickSort, но при слишком глубокой рекурсии переключается на HeapSort. Гарантирует `O(n log n)` даже в худшем случае. Используется в C++, Rust, Julia.
- **Shell Sort** *(Шелла)*
  Улучшение сортировки вставками: сначала сортирует элементы на большом расстоянии, потом уменьшает его. Быстрее вставками, но нестабильный и слабо предсказуем по производительности.

🧠 Алгоритмы сортировки, используемые в Julia

По умолчанию, функции `sort` и `sort!` в Julia 1.11 применяют **гибридную стратегию**, которая может включать следующие алгоритмы.

- **RadixSort** — для целочисленных и некоторых чисел с плавающей точкой.
- **CountingSort** — для массивов с ограниченным диапазоном значений.
- **QuickSort** и его модификации — для общих случаев.
- **InsertionSort** — для небольших массивов или почти отсортированных данных.

Этот подход обеспечивает высокую производительность и стабильность сортировки для большинства типов данных. Начиная с Julia 1.9, гарантируется стабильность сортировки по умолчанию, что означает сохранение относительного порядка равных элементов. docs.julialang.org