Содержание

Турнирная сортировка: Оценка сложности

Турнирная сортировка (Tournament Sort) — это алгоритм, который используется для упорядочивания элементов путём моделирования процесса турнира. В этом алгоритме элементы сравниваются парами, победитель каждой пары продолжает участие в следующем раунде, пока не останется только один элемент — максимальный (или минимальный, в зависимости от задачи). Затем процесс повторяется для оставшихся элементов.

Основные шаги алгоритма:

1. Построить бинарное дерево, представляющее турнир:
   - Листья дерева — это элементы исходного массива.
   - Внутренние узлы — это результаты сравнений (победители).
2. Выбрать наибольший элемент (или минимальный) — он будет в корне дерева.
3. Заменить этот элемент "пустым значением"(например, $-\infty$ для поиска максимального элемента или $+\infty$ для минимального).
4. Перестроить дерево для нахождения следующего победителя.
5. Повторять шаги 2–4 до тех пор, пока все элементы не будут упорядочены.

Оценка сложности

1. Построение турнирного дерева

- Для n элементов требуется $O(n)$ операций, чтобы построить дерево.
- Высота бинарного дерева составляет $\lceil \log_2 n \rceil$.

2. Поиск победителя

- Для нахождения максимального элемента требуется $O(\log n)$ сравнений (высота дерева).

3. Замена элемента и обновление дерева

- После удаления победителя дерево обновляется за $O(\log n)$.

Общая сложность

- Так как мы выполняем $O(n)$ итераций для нахождения каждого элемента, и каждая итерация включает $O(\log n)$ операций, общая временная сложность составляет:

$$O(n \log n)$$

Дополнительно

- Турнирная сортировка менее эффективна на практике, чем, например, **быстрая сортировка** или **сортировка слиянием**, из-за высокой константы перед логарифмом и необходимости работы с деревом.
- Этот метод полезен в случаях, когда данные поступают в потоковом режиме (*streaming data*), так как можно обновлять дерево динамически.

Понятие пирамиды. Построение пирамиды. Процедура просеивания и оценка сложности

Понятие пирамиды

Пирамида (Heap) — это полное бинарное дерево, удовлетворяющее следующему свойству:

- Значение в любом узле не меньше (или не больше) значений в его потомках.

В зависимости от свойства, различают два типа пирамид:

1. **Макс-куча (Max-Heap):** значение каждого узла $\geq$ значений его потомков.
2. **Мин-куча (Min-Heap):** значение каждого узла $\leq$ значений его потомков.

Пирамида обычно представляется в виде массива, где:

- Для узла с индексом i:

$$\begin{aligned} \text{Левый потомок:} &\ 2i+1, \\ \text{Правый потомок:} &\ 2i+2, \\ \text{Родительский узел:} &\ \lfloor (i-1)/2 \rfloor. \end{aligned}$$

Построение пирамиды

Для построения пирамиды используется метод **"восходящего построения"** (*build-heap*), который работает следующим образом:

1. Начинаем с последнего внутреннего узла массива (индекс $\lfloor n/2 \rfloor - 1$) и выполняем процедуру **просеивания вниз** (*heapify*).
2. Продолжаем процесс для всех узлов вплоть до корня (индекс 0).

Процедура просеивания

Просеивание (*heapify*) — это процесс перестановки элементов в пирамиде для восстановления её свойств. Рассмотрим случай **макс-кучи**:

1. Выбираем узел, который нарушает свойство пирамиды.
2. Сравниваем его с двумя потомками.
3. Если значение узла меньше значения его потомка, меняем их местами с наибольшим потомком.
4. Повторяем процесс для нового положения узла, пока свойство пирамиды не будет восстановлено.

Оценка сложности

1. Просеивание одного узла:

- Высота бинарного дерева составляет $h = \lfloor \log_2 n \rfloor$.
- Для восстановления свойств пирамиды требуется $O(h) = O(\log n)$ операций.

2. Построение пирамиды:

- Для каждого узла выполняется просеивание.
- Общая сложность:

$$\sum_{i=1}^{\lfloor n/2 \rfloor} O(\log i) \approx O(n).$$

Итог:

- Построение пирамиды выполняется за $O(n)$.
- Вставка и удаление элемента из пирамиды выполняются за $O(\log n)$.

Алгоритм пирамидальной сортировки

Пирамидальная сортировка (Heap Sort) — это алгоритм сортировки, основанный на использовании структуры данных **кучи** (*heap*). Основная идея заключается в преобразовании массива в кучу, а затем в последовательном извлечении максимального (или минимального) элемента для формирования отсортированного массива.

Основные шаги алгоритма

1. Построить **макс-кучу** из исходного массива:
   - Использовать метод **build-heap**.
2. Повторять следующие действия для сортировки:
   - Обменять корневой элемент (наибольший) с последним элементом кучи.
   - Уменьшить размер кучи на 1.
   - Восстановить свойства кучи с помощью **просеивания вниз** (*heapify*).
3. Повторять шаг 2 до тех пор, пока размер кучи не станет равным 1.

Псевдокод алгоритма

```
HeapSort(array):
    1. BuildMaxHeap(array)
    2. for i from length(array) - 1 down to 1:
        a. Swap(array[0], array[i])
        b. size -= 1
        c. Heapify(array, 0)
```

Процедура *Heapify*

Процедура *Heapify* выполняет восстановление свойств **макс-кучи**, начиная с узла, который может нарушать эти свойства.

```
Heapify(array, i):
    1. left = 2 * i + 1
    2. right = 2 * i + 2
    3. largest = i
    4. if left < size and array[left] > array[largest]:
        largest = left
    5. if right < size and array[right] > array[largest]:
        largest = right
    6. if largest != i:
        Swap(array[i], array[largest])
        Heapify(array, largest)
```

Оценка сложности

1. Построение кучи:

- Построение кучи выполняется за $O(n)$, так как просеивание вызывается для каждого узла, и сложность уменьшается на нижних уровнях дерева.

2. Извлечение максимального элемента и восстановление кучи:

- Каждое извлечение максимального элемента требует $O(\log n)$ операций (высота дерева).
- Для n элементов выполняется $O(n)$ операций извлечения.
- Общая сложность: $O(n \log n)$.

Итоговая сложность:

$$O(n) + O(n \log n) = O(n \log n)$$

Особенности алгоритма

- Пирамидальная сортировка является **неустойчивым** методом сортировки.
- Использует $O(1)$ дополнительной памяти, так как сортировка выполняется на месте.

Инвариант цикла

Инвариант цикла — это утверждение, которое выполняется при каждой итерации цикла. Для правильности алгоритма важно:

- Инвариант должен быть истинным перед началом выполнения цикла.
- Инвариант должен сохраняться на каждой итерации.
- После завершения цикла инвариант должен привести к утверждению о корректности результата.

Пример: Сортировка вставкой

Рассмотрим алгоритм сортировки вставкой. Его идея заключается в последовательной обработке элементов массива и вставке текущего элемента в правильное место в уже отсортированной части массива.

Алгоритм сортировки вставкой

Псевдокод алгоритма:

```
InsertionSort(array):
    for i from 1 to length(array) - 1:
        key = array[i]
        j = i - 1
        while j >= 0 and array[j] > key:
            array[j + 1] = array[j]
            j -= 1
        array[j + 1] = key
```

Инвариант цикла

Инвариант для внешнего цикла:

> После i-й итерации внешний цикл гарантирует, что первые i элементов массива (`array[0..i-1]`) упорядочены.

Доказательство инварианта

Докажем инвариант для внешнего цикла по трём шагам:

1. База: инвариант верен перед первой итерацией.

Перед началом работы внешнего цикла $i = 1$. Подмассив `array[0..i-1]` состоит из одного элемента (`array[0]`), и он тривиально упорядочен.

2. Сохранение: инвариант сохраняется на каждой итерации.

На i-й итерации алгоритм берёт элемент `key = array[i]` и вставляет его в нужное место в отсортированном подмассиве `array[0..i-1]`. После

завершения цикла `while` элемент `key` оказывается на своём месте, и подмассив `array[0..i]` становится упорядоченным.

3. Завершение: инвариант приводит к корректности алгоритма.

После завершения внешнего цикла $i = n$, где n — длина массива. Инвариант утверждает, что подмассив `array[0..n-1]` упорядочен. Это означает, что весь массив отсортирован.

Оценка сложности

- **Временная сложность:**
  - В худшем случае (массив отсортирован в обратном порядке) потребуется $O(n^2)$ операций.
  - В лучшем случае (массив уже отсортирован) потребуется $O(n)$ операций.
- **Дополнительная память:** $O(1)$.

Задача о найме

Описание задачи:

Компания нанимает кандидатов на вакансию, рассматривая их одного за другим в случайном порядке. На каждом шаге доступна информация только о текущем кандидате и уже просмотренных. Решение о найме принимается немедленно: новый кандидат принимается на работу, если его квалификация лучше всех предыдущих.

Алгоритм решения задачи

Псевдокод:

```
HireCandidate(candidates):
    best = None
    for candidate in candidates:
        if candidate > best:
            best = candidate
            Hire(candidate)
```

Здесь `Hire(candidate)` — процедура найма кандидата.

Анализ в худшем случае

В худшем случае каждый новый кандидат оказывается лучше всех предыдущих, и алгоритм принимает всех кандидатов. Если в списке n кандидатов, потребуется n наймов.

Временная сложность: $O(n)$.

Анализ в среднем случае

Пусть кандидаты поступают в случайном порядке. Вероятность того, что i-й кандидат окажется лучшим среди первых i кандидатов, равна $1/i$. Таким образом, матожидание числа наймов равно:

$$E[\text{число наймов}] = \sum_{i=1}^{n} \frac{1}{i}.$$

Эта сумма равна H_n (гармоническое число), которое аппроксимируется как:

$$H_n \approx \ln n + \gamma,$$

где γ — константа Эйлера-Маскерони.

Средняя сложность: $O(\ln n)$.

Итоговый анализ

- В худшем случае потребуется $O(n)$ наймов.
- В среднем случае потребуется $O(\ln n)$ наймов.

Понятие рандомизации алгоритма

Рандомизация в алгоритмах — это использование случайных чисел или случайных выборов для управления поведением алгоритма. Рандомизированные алгоритмы могут давать разные результаты или проходить разные пути выполнения на одном и том же входе.

Классификация рандомизированных алгоритмов

1. **Лас-Вегас алгоритмы:**

- Всегда возвращают корректный результат.
- Используют случайность, чтобы улучшить производительность или уменьшить время работы.
- Пример: быстрая сортировка с рандомизацией выбора опорного элемента.

2. **Монте-Карло алгоритмы:**

- Могут возвращать некорректный результат с небольшой вероятностью.
- Используют случайность для получения приближённых решений или ускорения вычислений.
- Пример: тест на простоту Миллера-Рабина.

Преимущества рандомизации

- Простота реализации.
- Избежание худших случаев, зависящих от структуры входных данных.
- Более высокая производительность в среднем случае.

Недостатки рандомизации

- Возможность некорректных решений (в случае Монте-Карло алгоритмов).
- Потребность в качественном источнике случайных чисел.

Пример рандомизированного алгоритма

Рассмотрим модификацию быстрой сортировки, в которой опорный элемент выбирается случайно:

```
RandomizedQuickSort(array, left, right):
    if left < right:
        pivot = Random(left, right)
        Swap(array[pivot], array[right])
        q = Partition(array, left, right)
        RandomizedQuickSort(array, left, q - 1)
        RandomizedQuickSort(array, q + 1, right)
```

Рандомизация выбора опорного элемента снижает вероятность того, что алгоритм попадёт в худший случай.

Заключение

Рандомизация — это мощный инструмент для улучшения производительности алгоритмов, особенно в задачах, где структура данных может приводить к худшим случаям для детерминированных алгоритмов.

Генерирование случайной перестановки

Описание задачи:

Задача состоит в том, чтобы преобразовать массив A длины n в случайную перестановку, такую что каждая перестановка имеет равную вероятность.

Алгоритм Фишера-Йетса

Псевдокод:

```
RandomPermutation(A):
    n = length(A)
    for i from n - 1 downto 1:
        j = Random(0, i)
        Swap(A[i], A[j])
```

Описание алгоритма:

- На i-й итерации выбирается случайный индекс j из диапазона $[0, i]$.
- Элементы $A[i]$ и $A[j]$ меняются местами.
- Таким образом, элемент $A[i]$ фиксируется, а остальные элементы продолжают перемешиваться.

Доказательство корректности через инвариант

Инвариант цикла: После завершения k-й итерации цикла:

- Элементы $A[k+1..n-1]$ фиксированы и являются случайной перестановкой.
- Элементы $A[0..k]$ перемешиваются равномерно.

Доказательство:

1. **База: инвариант верен перед началом цикла.**

Перед первой итерацией ($k = n - 1$), все элементы массива считаются перемешанными равномерно, так как никаких операций ещё не выполнено.

2. **Сохранение: инвариант сохраняется на каждой итерации.**

На k-й итерации алгоритм выбирает случайный индекс $j \in [0, k]$ и меняет местами $A[k]$ и $A[j]$. После этого:

- Элемент $A[k]$ фиксируется на случайной позиции, не зависящей от предыдущих перестановок.
- Остальные элементы $A[0..k-1]$ остаются равномерно перемешанными.

Таким образом, инвариант сохраняется.

3. **Завершение: инвариант приводит к корректности.**

После последней итерации ($k = 0$) массив A состоит из элементов, равномерно распределённых по всем возможным перестановкам.

Оценка сложности

- **Временная сложность:** $O(n)$, так как цикл выполняется n итераций, каждая из которых требует $O(1)$ операций.
- **Дополнительная память:** $O(1)$, так как алгоритм выполняется на месте.

Заключение

Алгоритм Фишера-Йетса эффективно генерирует случайные перестановки массива с равномерным распределением, используя случайные числа и операции обмена.

Быстрая сортировка. Разбиение Хоара

Быстрая сортировка (QuickSort) — это алгоритм "разделяй и властвуй который сортирует массив путём рекурсивного разбиения его на подмассивы. Одним из ключевых элементов алгоритма является процедура **разбиения**, которая разделяет массив на две части.

Разбиение Хоара

Метод разбиения Хоара выбирает опорный элемент (*pivot*) и переставляет элементы массива так, чтобы:

- Все элементы, меньшие или равные опорному, находились слева.
- Все элементы, большие опорного, находились справа.

Псевдокод разбиения Хоара

```
PartitionHoare(array, low, high):
    pivot = array[low]
    i = low - 1
    j = high + 1
    while true:
        repeat:
            j -= 1
        until array[j] <= pivot
        repeat:
            i += 1
        until array[i] >= pivot
        if i >= j:
            return j
        Swap(array[i], array[j])
```

Описание работы

1. Инициализируются два указателя: i (слева от массива) и j (справа от массива). 2. Указатель j перемещается влево, пока не найдётся элемент, меньший или равный опорному. 3. Указатель i перемещается вправо, пока не найдётся элемент, больший или равный опорному. 4. Если $i < j$, найденные элементы меняются местами, чтобы обеспечить корректное разбиение. 5. Если $i \geq j$, процедура завершается, возвращая индекс j.

Быстрая сортировка с разбиением Хоара

Псевдокод:

```
QuickSort(array, low, high):
    if low < high:
```

```
p = PartitionHoare(array, low, high)
QuickSort(array, low, p)
QuickSort(array, p + 1, high)
```

Оценка сложности

- **Средний случай:** $O(n \log n)$. Разбиение делит массив примерно пополам на каждой итерации.
- **Худший случай:** $O(n^2)$. Происходит, если массив уже отсортирован или опорный элемент каждый раз оказывается минимальным или максимальным.
- **Дополнительная память:** $O(\log n)$ для рекурсивного стека вызовов.

Особенности разбиения Хоара

- Требует меньше операций обмена по сравнению с другими методами разбиения.
- Индекс разбиения (j) не гарантирует, что опорный элемент находится на своей окончательной позиции.

Быстрая сортировка. Разбиение Ломуто

Разбиение Ломуто — это метод разбиения массива вокруг опорного элемента (*pivot*) для использования в быстрой сортировке.

Псевдокод разбиения Ломуто

```
PartitionLomuto(array, low, high):
    pivot = array[high]
    i = low - 1
    for j = low to high - 1:
        if array[j] <= pivot:
            i += 1
            Swap(array[i], array[j])
    Swap(array[i + 1], array[high])
    return i + 1
```

Описание работы

1. Выбирается опорный элемент (`pivot`) — последний элемент массива. 2. Все элементы, меньшие или равные `pivot`, перемещаются в левую часть массива. 3. Все элементы, большие `pivot`, остаются справа. 4. Опорный элемент помещается на свою окончательную позицию, и процедура возвращает её индекс.

Особенности разбиения Ломуто

- Проще в реализации, чем разбиение Хоара.
- Меняет местами больше элементов, чем метод Хоара.
- Подходит для учебных целей и небольших массивов.

Выбор опорного элемента при быстрой сортировке

Выбор опорного элемента (*pivot*) влияет на производительность быстрой сортировки:

- **Первый элемент массива:** Подходит для уже перемешанных массивов.
- **Последний элемент массива:** Часто используется для простоты реализации (например, в разбиении Ломуто).
- **Случайный элемент:** Уменьшает вероятность худшего случая; реализуется с помощью генератора случайных чисел.
- **Медиана трёх:** Выбирается медиана из первого, среднего и последнего элементов массива. Этот метод уменьшает вероятность плохого разбиения.

Слияние отсортированных массивов. Сложность слияния

Слияние двух отсортированных массивов — это процесс объединения массивов в один отсортированный массив.

Псевдокод слияния

```
Merge(array1, array2):
    result = []
    i = 0, j = 0
    while i < length(array1) and j < length(array2):
        if array1[i] <= array2[j]:
            Append(result, array1[i])
            i += 1
        else:
            Append(result, array2[j])
            j += 1
    Append remaining elements of array1 or array2 to result
    return result
```

Сложность слияния

- **Временная сложность:** $O(n)$, где $n = n_1 + n_2$ — суммарная длина массивов.
- **Дополнительная память:** $O(n)$ для хранения результирующего массива.

Рекурсивная сортировка слиянием. Оценка сложности

Сортировка слиянием (MergeSort) — это алгоритм "разделяй и властвуй который рекурсивно делит массив на подмассивы, сортирует их и сливает.

Псевдокод сортировки слиянием

```
MergeSort(array, left, right):
    if left < right:
        mid = (left + right) / 2
        MergeSort(array, left, mid)
        MergeSort(array, mid + 1, right)
        Merge(array, left, mid, right)
```

Оценка сложности

- **Временная сложность:** $O(n \log n)$, так как на каждом уровне рекурсии выполняется $O(n)$ слияний, а глубина рекурсии равна $O(\log n)$.
- **Дополнительная память:** $O(n)$ для хранения временного массива.

Распределительная сортировка (LSD). Анализ сложности

Сортировка по разрядам (LSD) — это алгоритм сортировки, который обрабатывает числа поразрядно, начиная с младших разрядов.

Псевдокод

```
LSDRadixSort(array, maxDigit):
    for d = 0 to maxDigit - 1:
        Sort array by digit d using stable sort
```

Анализ сложности

- **Временная сложность:** $O(n \cdot k)$, где n — количество элементов, k — количество разрядов.
- **Дополнительная память:** $O(n)$ для временного массива.

Карманная сортировка. Анализ сложности

Карманная сортировка (Bucket Sort) разделяет элементы массива на "карманы"(диапазоны значений) и сортирует каждый карман отдельно.

Анализ сложности

- **Лучший случай:** $O(n)$, если элементы равномерно распределены и карманы хорошо сбалансированы.
- **Худший случай:** $O(n^2)$, если все элементы попадают в один карман.

Карманная сортировка со сложностью $O(n \log n)$ в худшем случае

Для достижения $O(n \log n)$ в худшем случае:

- Карманы распределяются сбалансировано.
- Внутри каждого кармана используется сортировка слиянием или быстрая сортировка.

Алгоритм

1. Разделите массив на k карманов с равными диапазонами.
2. Сортируйте каждый карман с помощью сортировки слиянием.
3. Объедините элементы из всех карманов в порядке их сортировки.

Сложность

- **Временная сложность:** $O(n \log n)$, если размер каждого кармана ограничен.
- **Дополнительная память:** $O(n)$.

Порядковые статистики. Нахождение с линейной сложностью

Порядковая статистика k-го порядка массива A — это k-й по величине элемент массива.

Алгоритм нахождения с линейной сложностью

Алгоритм используется для нахождения k-й порядковой статистики с линейной временной сложностью $O(n)$. Это модификация алгоритма QuickSort — **Randomized-Select**.

```
RandomizedSelect(array, low, high, k):
    if low == high:
        return array[low]
    pivotIndex = RandomizedPartition(array, low, high)
    if k == pivotIndex:
        return array[k]
    elif k < pivotIndex:
        return RandomizedSelect(array, low, pivotIndex - 1, k)
    else:
        return RandomizedSelect(array, pivotIndex + 1, high, k)
```

Сложность: $O(n)$ в среднем случае.

Односторонний двоичный поиск на примере определения границы серий

Односторонний двоичный поиск используется для поиска границ в упорядоченных последовательностях.

Пример: определение границы серий

Задача: найти минимальный индекс k, такой что $A[k] > x$.

Алгоритм:

```
BinarySearchBoundary(array, x):
    left = 0
    right = len(array) - 1
    while left < right:
        mid = (left + right) // 2
        if array[mid] > x:
            right = mid
        else:
            left = mid + 1
    return left
```

Сложность: $O(\log n)$.

Задача определения пересечения множеств

Для нахождения пересечения двух массивов A и B: 1. Преобразуйте один из массивов в `set`. 2. Проверьте наличие элементов другого массива в этом `set`.

```
Intersection(A, B):
    setA = set(A)
    result = []
    for element in B:
        if element in setA:
            result.append(element)
    return result
```

Сложность: $O(n + m)$, где n и m — размеры массивов.

Определение дубликатов в массиве за $O(n \log n)$

Алгоритм

1. Отсортируйте массив. 2. Пройдите по массиву, сравнивая соседние элементы.

```
FindDuplicates(array):
    array.sort()
    for i in range(1, len(array)):
        if array[i] == array[i - 1]:
            return True
    return False
```

Сложность: $O(n \log n)$ из-за сортировки.

Задача определения частоты встречаемости элемента в массиве

Алгоритм

Используйте хеш-таблицу для подсчёта частот.

```
FrequencyCount(array):
    freq = {}
    for element in array:
        freq[element] = freq.get(element, 0) + 1
    return freq
```

Сложность: $O(n)$.

Понятие амортизационного анализа на примере операции MultiPop()

MultiPop(Stack, k): удаляет до k элементов из стека.

Анализ

- Операция `Pop()` имеет $O(1)$ сложность.
- За n операций `Push()` стек может быть полностью опустошён, выполняя $O(n)$ операций `Pop()`.
- **Амортизированная сложность:** $O(1)$ для одной операции, так как суммарные затраты делятся на общее количество операций.

Динамическая таблица (вектор). Амортизированная сложность расширения

Динамическая таблица (вектор) увеличивается при добавлении элементов, если текущая ёмкость исчерпана.

Амортизационный анализ

- Если ёмкость таблицы удваивается, добавление n элементов потребует $O(2n)$ операций копирования.
- **Амортизированная сложность:** $O(1)$ на добавление одного элемента.

Добавление, удаление, поиск элементов в таблице. Прямая адресация

Прямая адресация используется для работы с элементами, ключи которых принадлежат малому диапазону.

Операции

- **Добавление:** $O(1)$.
- **Удаление:** $O(1)$.
- **Поиск:** $O(1)$.

Пример

```
DirectAddressTable(size):
    table = [None] * size

Insert(table, key, value):
    table[key] = value

Delete(table, key):
    table[key] = None

Search(table, key):
    return table[key]
```

Понятие хэш-таблицы. Разрешение коллизий с помощью цепочек

Хэш-таблица — это структура данных, которая обеспечивает эффективное добавление, удаление и поиск элементов за $O(1)$ в среднем случае.

Разрешение коллизий с помощью цепочек

При коллизии элементы, хэшируемые в одну и ту же ячейку, хранятся в связном списке (*цепочке*). Операции производятся путем поиска в цепочке.

Операции:

- **Добавление:** Хэшируем ключ и добавляем элемент в соответствующую цепочку.
- **Поиск:** Хэшируем ключ, затем ищем элемент в цепочке.
- **Удаление:** Хэшируем ключ, удаляем элемент из цепочки.

Сложность:

- Средний случай: $O(1)$.
- Худший случай (все элементы в одной цепочке): $O(n)$.

Хэширование делением с остатком

Метод хэширования: $h(k) = k \bmod m$, где:

- k — ключ,
- m — размер хэш-таблицы.

Выбор m: m должно быть простым числом, не близким к степеням 2.

Хэширование умножением

Метод хэширования: $h(k) = \lfloor m(kA \bmod 1) \rfloor$, где:

- A — дробное число $0 < A < 1$,
- m — размер хэш-таблицы.

Выбор A: Обычно выбирают $A \approx \frac{\sqrt{5}-1}{2}$.

Универсальное хэширование

Универсальное хэширование выбирает случайную хэш-функцию из семейства функций, чтобы уменьшить вероятность коллизий.

Функция: $h_{a,b}(k) = ((ak + b) \bmod p) \bmod m$, где:

- a, b — случайные числа, $1 \leq a < p$, $0 \leq b < p$,
- p — простое число, большее максимального ключа.

Разрешение коллизий с помощью открытой адресации

Открытая адресация ищет другую ячейку таблицы для элемента при коллизии.

Методы поиска

- **Линейное исследование:** $h(i,k) = (h(k) + i) \bmod m$.
- **Двойное хэширование:** $h(i,k) = (h_1(k) + i \cdot h_2(k)) \bmod m$.

Сложность: $O(1)$ в среднем, $O(n)$ в худшем случае.

Определение ОЛРУ. Алгоритм поиска решения. Случай разных корней характеристического уравнения

Однородное линейное рекуррентное уравнение (ОЛРУ):

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k}.$$

Решение:

1. Найти характеристическое уравнение: $\lambda^k - c_1\lambda^{k-1} - \cdots - c_k = 0$.
2. Решить уравнение, найдя корни $\lambda_1, \lambda_2, \ldots, \lambda_k$.
3. Если корни различны, общее решение:

$$a_n = A_1\lambda_1^n + A_2\lambda_2^n + \cdots + A_k\lambda_k^n.$$

Определение ОЛРУ. Алгоритм поиска решения. Случай кратных корней характеристического уравнения

Если характеристическое уравнение имеет кратные корни λ, то решение принимает вид:

$$a_n = (A_1 + A_2 n + \cdots + A_m n^{m-1})\lambda^n,$$

где m — кратность корня.

Числа Фибоначчи. Определение, формула в замкнутом виде

Числа Фибоначчи определяются рекурсивно:

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2}.$$

Формула в замкнутом виде (формула Бине):

$$F_n = \frac{\varphi^n - \psi^n}{\sqrt{5}},$$

где:

$$\varphi = \frac{1+\sqrt{5}}{2}, \quad \psi = \frac{1-\sqrt{5}}{2}.$$

Определение НЛРУ. Общий алгоритм поиска решения

Неоднородное линейное рекуррентное уравнение (НЛРУ):

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k} + f(n),$$

где $f(n)$ — функция неоднородности.

Решение

1. Найти общее решение $a_n^{(h)}$ однородного уравнения.
2. Найти частное решение $a_n^{(p)}$ неоднородного уравнения.
3. Общее решение:
$$a_n = a_n^{(h)} + a_n^{(p)}.$$

Поиск частного решения НЛРУ при функции-константе

Уравнение:

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k} + C,$$

где C — константа.

Частное решение: Предположим, что $a_n^{(p)} = A$, где A — константа. Подставим в уравнение:

$$A = c_1 A + c_2 A + \cdots + c_k A + C.$$

Решение:

$$A = \frac{C}{1 - (c_1 + c_2 + \cdots + c_k)}, \quad \text{если } 1 - (c_1 + c_2 + \cdots + c_k) \neq 0.$$

Поиск частного решения НЛРУ при функции-многочлене

Уравнение:

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k} + P(n),$$

где $P(n)$ — многочлен степени d.

Частное решение: Предположим, что $a_n^{(p)} = Q(n)$, где $Q(n)$ — многочлен той же степени d. Подставим в уравнение и определим коэффициенты $Q(n)$.

Пример: Если $P(n) = an + b$, то $a_n^{(p)}$ также линейный многочлен вида $An + B$. После подстановки в исходное уравнение находим A и B.

Поиск частного решения НЛРУ при функции-экспоненте

Уравнение:

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k} + C\lambda^n,$$

где λ — константа.

Частное решение: Предположим, что $a_n^{(p)} = A\lambda^n$. Подставим в уравнение:

$$A\lambda^n = c_1 A\lambda^{n-1} + c_2 A\lambda^{n-2} + \cdots + c_k A\lambda^{n-k} + C\lambda^n.$$

После деления на λ^n:

$$A = \frac{C}{1 - (c_1\lambda^{-1} + c_2\lambda^{-2} + \cdots + c_k\lambda^{-k})}.$$

Решение рекуррентных уравнений подстановкой на примере $T(n) = aT(n/m) + bn, T(1) = b$

Рекуррентное уравнение:

$$T(n) = aT\left(\frac{n}{m}\right) + bn, \quad T(1) = b.$$

Решение методом подстановки:

1. Подставляем рекуррентное выражение:

$$T(n) = a\left[aT\left(\frac{n}{m^2}\right) + b\frac{n}{m}\right] + bn.$$

2. После k шагов:

$$T(n) = a^k T\left(\frac{n}{m^k}\right) + b\sum_{i=0}^{k-1} a^i \frac{n}{m^i}.$$

3. Когда $k = \log_m n$, достигаем базового случая $T(1) = b$:

$$T(n) = n^{\log_m a} + bn \sum_{i=0}^{\log_m n - 1} \left(\frac{a}{m}\right)^i.$$

Если $a = m$, $T(n) = O(n \log n)$.

«Угадывание» итогового решения

Метод «угадывания» состоит из:

1. Предположения о виде решения.
2. Подстановки предположения в уравнение.
3. Доказательства предположения методом математической индукции.

Рекуррентные соотношения: основная теорема. Формулировка

Основная теорема: Рассмотрим рекуррентное соотношение:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n),$$

где $a \geq 1$, $b > 1$, $f(n)$ — асимптотически положительная функция.

Решение:

- Если $f(n) = O(n^{\log_b a - \epsilon})$, то $T(n) = \Theta(n^{\log_b a})$.
- Если $f(n) = \Theta(n^{\log_b a} \log^k n)$, то $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
- Если $f(n) = \Omega(n^{\log_b a + \epsilon})$ и выполняется условие «доминирования», то $T(n) = \Theta(f(n))$.

Оценка сумм через интеграл. Основная идея, оценки сверху и снизу

Идея: Суммы вида $\sum_{i=1}^{n} f(i)$ можно оценить с помощью интеграла:

$$\int_1^n f(x)\,dx \le \sum_{i=1}^{n} f(i) \le \int_1^{n+1} f(x)\,dx.$$

Пример оценки сверху

Если $f(i) = \frac{1}{i}$, то:

$$\sum_{i=1}^{n} \frac{1}{i} \le \int_1^{n+1} \frac{1}{x}\,dx = \ln(n+1).$$

Пример оценки снизу

Для того же $f(i)$:

$$\sum_{i=1}^{n} \frac{1}{i} \ge \int_1^{n} \frac{1}{x}\,dx = \ln(n).$$

Таким образом, $\sum_{i=1}^{n} \frac{1}{i} = \Theta(\log n)$.