Конспект лекций Михаила Густокашина по дисциплине «Алгоритмы и структуры данных 1»

От добросовестной слушательницы Ли Людмилы БПМИИ247

2024 год

1 Алгоритмы и их сложность

Обращение к любой ячейке памяти занимает одинаковое количество времени, а процессоры мы рассматриваем одноядерные. $f(x) = O(g(x))$ - время работы алгоритма (оно не измеряется в секундах). Константа не влияет, если вы увеличиваете размер входных данных: $O(3n^2) = O(n^2)$. Асимптотическая оценка нужна, чтобы проверить во сколько раз замедляется программа при увеличении входных данных. Она проводится сверху, поэтому может быть неточной.

$\theta(f(x)) = g(x)$ - функции растут одинаково, прям вообще одинаково, то есть $O(f(x)) = O(g(x))$ и $O(g(x)) = O(f(x))$. А когда вы в реальной жизни будете писать алгоритм, там большую роль будет играть средний случай, но рассуждать надо в рамках худшего случая. В качестве примера рассмотрим поиск цикла отрицательного веса с помощью алгоритма Флойда: нам нужно считать квадратную таблицу и пробежаться по ней тремя циклами, в котором мы перебираем вершины. Считать одну переменную занимает O(1), что, кстати, не всегда правда, потому что если число очень большое, то это заняло бы O(кол-во разрядов числа), но у нас скорее всего число короткое. Код состоит из кусочков разной сложности, поэтому, чтобы оценить сложность алгоритма, мы выбираем самый худший случай (стоит повторить свойства о-символики). Чтобы понять, эффективен ли алгоритм, очень важно посчитать O большое, чтобы понять, не выходит ли вывод данных за рамки дозволенного. Если выходит, то нужно придумывать другое решение.

1.1 Основные ошибки и их причины.

TL - скорее всего появился вечный цикл или алгоритм очень криво написан и программа вырубилась, а также проблема может быть в том, что print в Python долго работает. а также медленные решения следует оборачивать в функции.
WA - ошибка в выводе, код неправильный. следует проверять на минимальные и максимальные вхождения элементов, чтобы избежать такого вердикта.
ML - алгоритм написан криво
Очень важно не писать код большой простынёй, то есть необходимо разделять на функции, чтобы знать, где есть баг, чтобы с лёгкостью его устранить при наличии.

2 Динамический массив. Стек. Очередь. Дек

Сложность следующих действий представлена внизу:
1. Чтение, вставка и удаление элементов в массиве выполняется за O(1), O(N) и O(N) соответственно.
2. Чтение, вставка и удаление элементов в связном списке выполняется за O(N), O(1) и O(1) соответственно.
Амортизатор - это устройство, которое выполняет функцию сглаживание колебаний.

Вот некоторые замечания по поводу работы с массивами и списками взятые из первого издания книги «Грокаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих» Адитья Бхаргава, 2022 г. (в дополнение к лекции):
1. В массиве все элементы хранятся в памяти рядом друг с другом;
2. В связном списке элементы распределяются в произвольных местах памяти, при этом в каждом элементе хранится адрес следующего элемента;
3. Массивы обеспечивают быстрое чтение;
4. Связные списки обеспечивают быструю вставку и удаление;
5. Все элементы массива должны быть однотипными (только int, только float и т.д.)

2.1 Банковский метод

Воспользуемся так называемым банковским методом, чтобы посчитать сложность алгоритма: вы говорите я буду брать с пользователя 3 монетки за операцию добавь в конец массива». При этом я буду платить исполнителю массива 1 монетку за присваивание, а также тратить их на само действие. Банковский метод можно использовать в следующем смысле: вы будете спрашивать с пользователя кол-во денег, и когда вам понадобиться сделать много операций (например, расширение, перестроение или что-то такое в этом духе), у вас должно быть в банке достаточное кол-во монеток накопленных, и тогда можно доказать сложность алгоритма. Если кол-во монеток никак не зависит от N, то есть константы, то сложность алгоритма будет линейная. Если бы мы просили у пользователя logN монеток, то сложность была бы O(NlogN) монеток.

2.2 Очередь

Как организована очередь (queue)? Возникает она довольно часто. Вот у нас есть бюро выдачи пропусков, в очереди стоят студенты. У нас есть первое место, но это неинтересно, поэтому, условно, взяв очередь из пяти человек, если подойдёт еще один, то есть шестой, мы сдвинем указатель конца очереди на него, и теперь концом очереди будет шестой человек, а не пятый, как до этого. Когда пропуск будет выдан, мы начнём переставлять студентов одного за другим, передвигая указатель конца очереди. Сложность добавления элементов в очередь это O(1): мы записываем число и увеличиваем переменную на 1. Сложность извлечения - O(N), где N - длина очереди, потому что мы каждый элемент вручную переставляем. Это плохо. Поэтому, так как студенты не двигаются в очереди, потому что не реагируют на внешние раздражители, то мы сдвигаем теперь не только указатель на конец очереди, но и указатель на начало очереди, чтобы не передвигать все элементы. И теперь извлечение элемента из начала очереди стоит O(1) (просто подвинуть элемент).

Предположим, за день прошла очередь из 1000 человек (примерно от покровки до метро курская). И к середине дня очередь будет стоять где-то в подземном переходе. Не отходя от нашей абстракции, это будет огромный массив, в котором куча пустого места. Чтобы решить данную проблему, мы обернём нашу очередь вокруг покры. Это называется кольцевой буфер. Нам понадобиться знание максимального кол-ва элементов k находящихся в очереди. Мы берем значения индекса начала и конца очереди по модулю k, и всё. Основная идея: кольцевой буфер с остатком от деления. Если индексы начала и конца очереди совпали, то необходимо применить динамическое расширение. Теперь применим динамическое сужение массива: если образовалось много пустого места в очереди, мы можем просто заново выделить меньший кусок памяти и скопировать туда нашу очередь. Операция pop() так же работает за O(1), так как сдвигаем size на один влево. Но есть мы будем неудачно использовать pop() и append(), то операции будут стоит нам уже O(N). Как избежать этого? Сужать массив не тогда, когда заполнена 1/2 массива, а 1/4. Писать код в таком случае нужно аккуратно.

2.3 Стек

Как и очередь, стек является абстрактной структурой данных. Ее можно реализовать с помощью связных списков, массивов, в общем, мно-

го разных способов. Но у нее есть фактически три операции: добавь в конец, забери начало и узнай размер. Обратиться к какому-то элементу в очереди теоретически возможно, но есть у вас односвязный список ничего не получится. Так вот, и казалось бы, зачем она нам? Для простоты операций. В отличие от очереди, стек представляет собой last in first out (LIFO). Представьте стопку книг, и всё что мы можем сделать: посмотреть какая книга сверху, заберём книгу сверху и добавим сверху еще одну. Динамически расширяемый массив отлично с этим справляется: есть операции push() и pop() для стека (как append() и pop() для массива) и top(), чтобы узнать последний элемент массива. Зачем нам такая дурацкая структура? Оказывается, она очень полезна. Вызываете вы функцию. Неважно, рекурсивная или нет. Произошёл вызов функции, а теперь, пользуясь стеком, пробегаемся по элементам, запоминая их значение, выделяя область памяти. По окончанию стека, функция останавливает свою работу. В качестве примера функции можно использовать функцию для подсчёта факториала числа. Так работает любой вызов, включая рекурсивный. То есть выделяется место для локальных переменных, мы запоминанием, куда подставлять значения. Во время работы действия выполняет у нас последняя вызванная функция, а все остальные ждут. Поэтому всегда, кроме самой верхней, приостановлены и ждут результата верхней. Это было первое применение стека.

Задача о правильной скобочной последовательности. Рассмотрим задачу про правильную скобочную последовательность. Например, такая последовательность (()()) является правильной, а ()))(- нет. Нам необходимо в задаче понять, является ли введённая последовательность правильной. При открывающихся скобках мы добавляем 1 в счётчик, а при закрывающейся отнимаем. Поэтому есть два критерия правильной скобочной последовательности: если баланс в конце не 0 и баланс в процессе не был отрицательным числом. Теперь добавим еще один тип скобок - квадратные. Но для такой последовательности ([)] концепт баланса не будет работать. Теперь мы используем в стек! Если встретилась открывающаяся скобка, то кладём в стек. Затем, когда встречается закрывающаяся скобка, мы проверяем, есть ли в стеке что-то, если есть, то смотрим, нужная ли нам скобка лежит сверху. Если нет, то скобочная последовательность неправильная, а если да, то удаляем сверху лежащую скобку. И так продолжаем до конца последовательности.

2.4 Разные записи выражений

Посмотрим на инфиксную запись $2 + (3 * 2 + 4) * 5$. В математике мы пользуемся префиксной записью: сначала операция, затем операнд (число). Пример: $sin(x)$. Существует постфиксная запись: сначала операнд, а затем операция. Последние две намного лучше остальных. Научимся ее читать! 2 3 2 * 4 + 5 * + и в ней нет скобок. Как это считать? Встретили операнд - положили в стек. Увидели операцию - взяли два элемента из стека, выполнили ее и положили обратно в стек. И так до конца выражения. Сейчас мы из инфиксной переведём в постфиксную. Как это делать? Существует некоторые правила. Операнд сразу идёт в ответ, видим операцию - кладём ее в стек, видим скобку открывающуюся - кладём в стек. Перед тем, как операция ляжет в стек, она выталкивает все предшествующие операции с приоритетом большим или равным, чем у нее. На картинке как раз таки представлены приоритеты операторов. Когда мы встречаем закрывающуюся скобку, то мы выталкиваем из стека всё, что было после открывающейся скобки, включая её саму.

Задача о домах и баннере ФКН. Даны здания разной высоты, и нам необходимо завесить людям окна, то есть расположить наш баннер ФКН так, чтобы он заполнял наибольшую площадь. Решить эту задачу довольно просто за $O(N^3)$, перебирая правую и левую границы, где мы будем весить баннер, и между ними просто найти самый низкий дом. Этот алгоритм легко оптимизировать до $O(N^2)$, но до линейного времени это сделать довольно сложно. Делать мы это будем так: давайте мы будем перебирать только 1 указатель, условно i, то есть номер дома, в который баннер упирается своей верхней границей. То есть баннер будет такой же высоты, что и этот дом. Насколько баннер может простираться влево и вправо? До ближайшего более низкого дома или до конца улицы. Вправо и влево. Но это всё еще $O(N^2)$, поэтому нам нужно подсчитать для каждого дома ближайший справа и слева меньший.

Давайте возьмём числа 6 4 9 7 6 7 9 3 соответственно количеству этажей в домах. Мы будем хранить в стеке те, для которых не нашли меньше. Я встречаю дом 6, кладу в стек, дальше стоит дом 4 и так как он меньше предыдущего, удаляю 6 из стека и кладу туда 4. Дальше встречаем дом 9, он не меньше предыдущего, значит просто кладём в стек, далее стоит домик 7 и он меньше 9, но больше 4, поэтому мы выкидываем 9 и кладём на её место 7. Концепция проста: если сверху лежащий элемент в стеке больше следующего в списке, то удаляем его, а если меньше - добавляем элемент из списка в стек. Таким образом, когда дойдём дома высотой

Операторы	Описание
`()`	Скобки
`**`	Возведение в степень
`+x`, `-x`, `~x`	Унарные плюс, минус и битовое отрицание
`*`, `/`, `//`, `%`	Умножение, деление, целочисленное деление, остаток от деления
`+`, `-`	Сложение и вычитание
`<<`, `>>`	Битовые сдвиги
`&`	Битовое *И*
`^`	Битовое исключающее *ИЛИ* (*XOR*)
`|`	Битовое *ИЛИ*
`==`, `!=`, `>`, `>=`, `<`, `<=`, `is`, `is not`, `in`, `not in`	Сравнение, проверка идентичности, проверка вхождения
`not`	Логическое *НЕ*
`and`	Логическое *И*
`or`	Логическое *ИЛИ*

3 этажа, он снесёт все дома в стеке. Следовательно, первый элемент в стеке это и есть ограничивающий нас дом. Данный алгоритм работает за O(N).

2.5 Дек

Это очередь с двумя концами. Рассмотрим задачу поиска минимума в окне. Пусть дана последовательность 3 2 5 4 5 6 2 7. Окно у нас размера $k = 3$. В лоб это решает за O(NK). Для умных ребят сложность этой задачи будет составлять O(NlogK). Сейчас мы научимся считать это за O(N). Используем дек.

УТВЕРЖДЕНИЕ: двойка встречается после тройки, а значит она будет во всех окнах содержащих тройку, а значит тройка уже никогда не

сможет стать минимумом.

Рассмотрим наше окно: кладём в стек 3, далее идёт 2, кладём ее и стираем 3, далее кладём 5, но не стираем 2, т.к. пятёрка больше 2 и так далее. Ответом, то есть минимумом первом окне, это самый первый элемент в стеке. Сдвигаем окно, и тройка выпадает из окна и приходит четвёрка. Кладём ее в стек и стираем пятёрку, так как 4 меньше 5. Двойка снова стала минимумом в окне, выводим ее (первый элемент стека). Теперь снова сдвигаем окно. И в этот момент уже двойка выпадает с окна, поэтому мы ее стираем из стека. Остаётся четвёрка. Дальше добавляем и удаляем по такому же принципу элементы. Таким образом действуем до конца строки.

3 Сортировки

3.1 Сортировка пузырьком

Начнём с простых сортировок. Самая известная: сортировка пузырьком. Идея: мы смотрим на пары соседних элементов, то есть у нас есть последовательность, состоящая из N элементов и мы смотрим на пары соседних элементов. Если они стоят не в правильном порядке, то меняем их местами. И так до конца строки. Как работает? Лёгкие идут наверх, а тяжелые идут на дно. Если за один такой проход, один элемент становится на своё место, то мы делаем N или $N-1$ операций, потому что если остался один элемент, то и так стоит на своём месте. Сложность: $O(N^2)$. Такой алгоритм потребляет немного дополнительной памяти, так как нам нужен внутренний счётчик, который бегает по элементам. Она смотрит только на соседние элементы. Когда мы к данным обращаемся подряд, это намного быстрее, чем если бы мы тыкали в случайный. Почему? Во-первых, хеш-процессоры, куда у вас прогружается 4 килобайта. Во-вторых, вспомните, как вычисляется место памяти для i-го элемента. Поэтому обращаться подряд быстрее, чем произвольно.

3.2 Сортировка выбором

Давайте реализуем сортировку выбором минимума. Пусть у нас есть отсортированная часть массива. Как нам сделать эту отсортированную часть массива на один больше? Нам нужно в оставшейся, не отсортированной части, найти минимум, и поменять местами с элементом, который идёт сразу после выделенной отсортированной подпоследовательности. Мы делаем N или $N-1$ операций. Она является самой быстрой из всех сортировок на больших данных.

3.3 Сортировка вставками

Начало такое: есть начало массива (отсортированный кусочек), оставшаяся часть и элемент, который идёт сразу после отсортированного кусочка массива. Наша цель: добавить его к отсортированной части. Как это сделать? Мы будем вставлять его в положенное ему место. То есть если x должен оказаться в каком-либо месте, где находится число больше x. Поэтому мы будет сдвигать это всё с правого края, вставим его и всё будет хорошо. Можно сразу как в сортировке пузырьком менять с соседями. Но при таком обмене кол-во присваиваний будет больше. В худшем случае будет $O(N^2)$.

3.4 Устойчивая сортировка

Устойчивая сортировка - это сортировка, которая сохраняет порядок элементов с одинаковыми ключами. Что это значит? Для обычных чисел это не имеет какого-либо смысла, но для каких-то структур не все поля участвуют в сравнении.

Например, на уроке физкультуры нужно выстроиться по росту. Допустим, есть Вася с ростом 175, еще какие-то люди до него, еще есть Федя с ростом 175 и т.д. Неважно как они сортировались. Но сортировка называется устойчивой, если она сохраняется взаимный порядок элементов с равными ключами. Ключом является рост. И если сортировка оставит Васю раньше Феди, то она устойчивая в любом случае. А если мы придумаем алгоритм, который каким-то образом поставит Федю раньше Васи, то такая сортировка является неустойчивой.
Давайте подумаем, как сделать такие сортировки устойчивыми. Это всегда можно сделать за $O(N)$ дополнительной памяти, просто добавив поле позиции в исходном массиве, но это не спортивно. Мы еще к этому вернёмся. Поэтому если элементы не равны, мы просто их не меняем. Но если мы сделаем их устойчивыми, то заплатим за это присваиваниями. Когда мы вызываем функцию sorted(), начинает работать быстрая сортировка (о ней в следующей лекции), а на маленьких данных используется сортировка выбором, потому что мы может себе это позволить.

Что такое количество информации? Это вопросы с ответом «Да» или «Нет», которые нужно задать, чтобы понять что-то про объект. Вопросы могут быть абсолютно любые. Что такое $log(N!)$? $log(N!) = log(1 \times 2 \times 3 \times ... \times N/2 \times ... \times N) \approx NlogN$. Это логарифм от произведения чисел. Чтобы доказать, что они асимптотически одинаковы, нам нужно сделать оценку сверху и снизу. Имейте в виду, везде фигурирует о-символика.
1) $log(N!) \leq NlogN$: очевидно, да, потому что мы повторяем N раз самый большой элемент.
2) $log(N!) \geq NlogN$: Эта часть сложнее, сейчас мы будем ее доказывать. В выражении $log(N/2) + ... + log(N)$ у нас $N/2$ слагаемых. Заменим это всё и скажем, что эта штука $\geq (N/2)log(N/2)$ (я N/2 раз повторил самое маленькое слагаемое). А константы роли не играют, поэтому $\geq NlogN$. Поэтому самая быстрая сортировка, отвечающая на вопросы «Да» и «Нет», которую мы может написать, имеет сложность $O(NlogN)$.

3.5 Сортировка слиянием

Пусть у вас есть две отсорченные кучки разлетевшихся конспектов. И вы хотите их объединить, чтобы новая куча тоже стала отсортированной. Как это сделать? Посмотреть номер верхнего листа, и где номер меньше, оттуда и берём. Когда мы имеем две отсортированные кучки, это работает. Какой у них должен быть размер? Они должны быть примерно равны. Давайте напишем такую функцию, но так, чтобы вы не могли ее использовать.

```
merge(seq1, seq2)
    ans = []
    p1 = 0
    p2 = 0
    while len(seq1) > p1 and len(seq2) < p2:
        if len(seq1) > p1 and (len(seq2) <= p2 or seq1[p1] <= seq2[p2]):
           ans.append(seq1[p1])
           p1 +=1
```

Очень важно аккуратно писать, иначе работать не будет. Очень важная вещь: логические выражения вычисляются линейно. Но код сверху довольно проблемный, поэтому использовать его не нужно. Это было бессмысленно и имело огромное количество ошибок, поэтому пишем подругому. Давайте схематично напишем, как работает такая сортировка.

```
mergesort(L, R):
    mergesort(L, (R + L)//2)
    mergesort((R + L)//2, R)
    merge(...)
```

То есть мы отсортировали, слили и показали. Но есть проблема: наш merge принимал на вход две последовательности, а не три. Нам придётся выделить память и скопировать ее, потому что мы не сможем сразу вставить в ответ. Поэтому мы будем разбивать массив на части и сортировать их. Это займёт O(logN). На каждом этапе такого деления мы пробегаем по элементам этих частей и сразу его определяем куда надо. Сложность получается O(NlogN). Если вы хотите, чтобы такая сортировка была устойчивой, берите нестрогое неравенство. Такая сортировка очень удобна для сортировки файлов.

3.6 Инверсии

Сколько обменов совершит сортировка пузырьком? Мы сортим и считаем количество инверсий между двумя указателями, а далее сдвигаем

их.

УТВЕРЖДЕНИЕ: можно расширить эту мысль до mergesort. Мы скажем: если мы отдельно для каждой из половинок до этого такую же операцию, то количество инверсий будет правильно подсчитано.

Допустим мы делали это для левой половинки и сделали ее отсорченной. Что нам нужно доказать? Что никакую инверсию мы не пропустили и никакую не подсчитали дважды. Пусть два элемента стояли неправильно и лежали в пределах этой половинки. Они здесь будут посчитаны в этой половинке? Их инверсия будет посчитана на этапы слияния. На следующем и возможно каких-то дальнейших шагах эта инверсия считаться не будет. Как проверить, что мы ни одной инверсии не пропустили? Допустим что существует пара чисел a и b для которых наш алгоритм инверсию не посчитал. Как нам прийти к противоречию? То есть найдётся такой разрез, что они находятся в разных половинках. Это единственный такой случай, потому что потом они окажутся в одной части. А так как наш алгоритм корректен, у нас эта инверсия засчитается. Мы посчитали все инверсии какие есть, и ни одну не пропустили.

3.7 Таблица

Лекция подошла к концу, поэтому прикрепляю таблицу с информацией обо всех перечисленных на ней сортировках!

—	Сложность	Потребление памяти	Присваивания	Устойчивость
Пузырёк	N^2	1	N^2	yes
Выбором	N^2	1	N	no
Вставками	N^2	1	N^2	yes
Слиянием	$NlogN$	N	$NlogN$	yes

4 Куча. Быстрая сортировка. Двоичный поиск

4.1 Бинарный поиск

Идея поиска заключается в том, чтобы брать элемент посередине, между границами, и сравнивать его с искомым. Если искомое больше(в случае правостороннего — не меньше), чем элемент сравнения, то сужаем область поиска так, чтобы новая левая граница была равна индексу середины предыдущей области. В противном случае присваиваем это значение правой границе.

Задача о попечительском совете. Рассмотрим задачу. Есть попечительский совет, в нем N человек из них K родителей. Требуется чтобы кол-во родителей составляла не менее трети от числа попечительского совета. Сколько родителей нужно добавить, чтобы их было не менее трети? Многие, решая эту задачу, облажались, так как решали задачу уравнением $(K + M)/N \geq 1/3$. Оно неверно, потому что делить надо $N + M$. Мы будем поступать следующим образом. Давайте мы будет подставлять какой-то конкретное M и будем смотреть, а не получилась ли у нас такая ситуация что у нас уже треть родителей или не треть. Понятное дело, пока мы добавили слишком мало, всё еще мало, но если мы добавим еще меньше, то процентное содержание будет еще меньше. Мы добавим родителей, но если мы добавим их еще больше, то плохо от этого никогда не станет. То есть наша функция имеет такой вид: сначала всё плохо, потом всё хорошо. Если наша функция имеет такой вид, то мы можем воспользоваться бинарным поиском.

Обозначим границы: L = 0, R = N. Даже если родителей будет половина, то это тоже хорошо. Это заведомо правильный ответ: как можно меньше думать. Дальше возможны нюансы. Я вам расскажу один из видов бинпоиска, потом поговорим как изобрести свой и убедиться, что он работает. В моем бинарном поиске граница поиска L и R будут включительно, то я от L до R и где-то там находится ответ. Самое маленькое такое число, что родителей будет достаточно. И поэтому меняем условие того, что я нашёл свой отрезок, нашел отрезок единичной длины, будет формулироваться так: while L $<$ R. Я посчитаю серединку этого отрезка и посмотрю, всё ли тут у меня хорошо. Если нет, то какую границу поиска я буду менять? Левую. Ну как бы недостаточно родителей я набрал, значит левее от L до M точно нет правильного ответа, и ответ где-то от M

до R. И я смело могу заметить левую границу на M. А если всё хорошо, и я попал в то место, что родителей хватает, тогда я буду менять правую границу поиска, потому что мне надо найти самое маленькое число. Самое главное, чтобы между L и R было какое-то значение. Мы выбрали какое-то, но оно не самое лучшее, поэтому мы меняем границу поиска. Опишем это в коде:

```
L = 0, R = N
while L < R:
    M = (L + R) // 2
    if good(M):
        R = M
    else:
        L = M + 1
```

Вы можете придумать свой бинпоиск. Но любой бинпоиск, который вы изобрели нужно проверять на двух элементах. Как выглядит функция good() в этой задаче? Функция good() принимает на вход параметр M. Она должна возвращать хорошо ли всё или плохо. Как это сделать? Смотрите, здесь конкретно в этой задаче есть деление. Это очень плохая операция, потому что она уходит в вещественные числа и там какая-то неприятность может случиться. Поэтому мы деление заменяем на умножение там где это можно. И получается, что это выражение приобретает вот такой вид:

```
good(M):
    return 3(K+M) >= N+M
```

Но не стоит забывать, что не каждую задачу бинпоиска можно решить формулой, как в этом случае.

Задача о задачах Васи. Васе нужно решить N задач. В первый день он решает одну задачу, во второй две, в третий день три задачи и так далее. Вопрос: сколько дней уйдет у Васи, чтобы решить хотя бы N задач в сумме? Решим бинпоиском. Смотрите, пока у Васи всё плохо, но будет хорошо, если он решил больше или равно N задач. Начиная с какого-то дня он решит эти задачи и продолжит решать дальше. Раз функция имеет такой вид, значит, мы может воспользоваться бинарным поиском. Сам бинарный поиск можем не менять, менять нужно границы, но так происходит не всегда: в зависимости от задач. Что нужно еще поменять? Функцию.

```
good(M):
    return M(M+1) >= 2N
```

Задача решена. Сложность бинарного поиска $O(log)$. Единственная проблема, связанная с границами, - это переполнения. Потому что если у вас не язык Python, то $M(M+1) >= 2N$ может переполниться. Будьте аккуратны.

Задача о стикерах Васи. Еще одна задачка. У Васи есть прямоугольная доска размером $N \times N$. Он на ней хочет разместить стикеры с напоминанием. Всего их K штук размером $M \times M$. Надо найти наибольшее M, чтобы квадратики не пересекались. Мы ищем последнее хорошее значение. Если сейчас всё хорошо, то какую границу мы должны менять? Левую. На что? На M. Если всё плохо, то меняем правую границу На M - 1. Иначе оно съедет за значение индексов. А также необходимо сделать округление вверх, чтобы не прийти к вечному циклу:

```
L = 0, R = N
while L < R:
    M = (L + R + 1) // 2
    if good(M):
        L = M
    else:
        R = M - 1
```

Также существует вещественный бинпоиск. С помощью него можно найти, например, процент по кредиту. А также можно воспользоваться двоичным поиском по производной. Если вы хотите найти минимум функции, обладающей свойством, что производная сначала отрицательная, а потом положительная, вы можете взять производную этой функции по определению, более того, производная вам не нужна - вам нужен знак производной. Поэтому делить необязательно, можно найти ту точку, где производная обратилась в ноль.

4.2 Куча. Heap

Она представляет из себя бинарное дерево. В куче есть свойстве: каждый родитель меньше (или больше) своих детей (в зависимости от того, какую кучу вы строите). Где же минимальный элемент? В корне дерева, то есть в самом низу. Давайте научимся добавлять элементы в кучу. Добавляем в свободное место. Если сын меньше отца, то меняем их местами по необходимости. Если условие отцов и детей всё ещё не выполнено, то

протакливаем наш элемент наверх. Кстати, если при добавлении ряд был заполнен, то ставим новый элемент в новый ряд.
Просеивание вверх - это сравнение элементов с его предком и swap их, если условие отцов и детей нарушено. Сложность такой операции добавление O(logN) в худшем случае. А теперь удаляем минимальный элемент из кучи. Мы его удаляем и на его место записываем самый последний элемент. Свойство кучи, естественно нарушилось, поэтому swapаем нас и детей, чтобы восстановить порядок (выбирать необходимо меньшего из детей, если их двое). Повторять эту операцию необходимо, пока всё не станет хорошо. Чтобы написать алгоритм для кучи, необходимо пронумеровать все его элементы сверху вниз, слева направо, а дальше создать массив с такой индексацией, и просеивать вверх по принципу отцов и детей.

```
def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
    if l < n and arr[i] < arr[l]:
        largest = l
    if r < n and arr[largest] < arr[r]:
        largest = r
    if largest != i:
        (arr[i], arr[largest]) = (arr[largest], arr[i])
        heapify(arr, n, largest)

def heapSort(arr):
    n = len(arr)
    for i in range(n // 2, -1, -1):
        heapify(arr, n, i)
    for i in range(n - 1, 0, -1):
        (arr[i], arr[0]) = (arr[0], arr[i])
        heapify(arr, i, 0)

arr = [12, 11, 13, 5, 6, 7, ]
heapSort(arr)
n = len(arr)
print('Sorted array is')
for i in range(n):
    print(arr[i])
```

4.3 Пирамидальная сортировка. Heapsort

Такая сортировка будет работать за NlogN. Но это отстойный способ. Можно сделать один трюк, чтобы не использовать дополнительную память. Пусть у меня есть некий массив, который нужно отсортить по возрастанию, стандартным образом: 7 3 10 8 2 2 1 5. Я глобально добиваюсь того, что у меня куча и отсорченная часть массива жили в одном массиве. Куча будет постепенно уменьшаться, а отсорченная часть массива будет постепенно расти.

Это идея не новая: она знакома вам с сортировки выбором минимума. Первое, что мы должны сделать, это превратить наш массив в кучу и получается, что это куча максимума, потому что у нас отсорченная часть массива будет жить в правой части нашего массива и соответственно первое, что мы должны поставить - это максимальный элемент. Давайте научимся превращать нашу кучу в кучу максимума. Давайте нарисуем кучу под 8 элементов. Располагаем их в дереве соответственно их индексам. И воссоздаем их правильный порядок. Сколько по времени занимает создать кучу? Ее можно построить за линейное время.

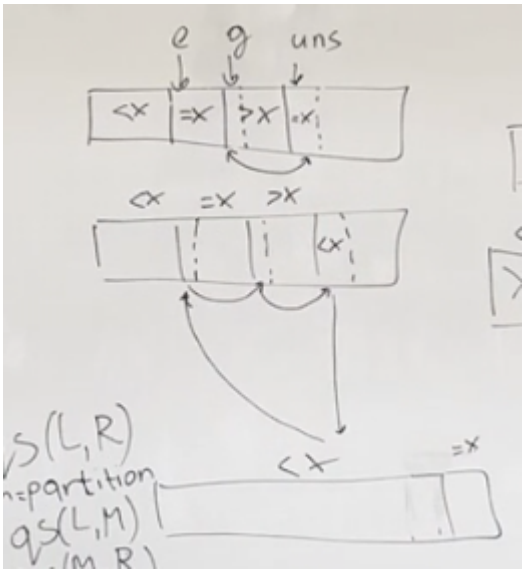
5 Сортировка подсчетом и поразрядная. Хеш-таблицы.

5.1 Быстрая сортировка

Начнём с быстрой сортировки. Иногда ее называют сортировкой Хоара. Идея: у нас есть некий массив, мы выбираем элемент, некоторое число (скорее всего элемент массива, но, в общем говоря, необязательно). И теперь мы добиваемся того эффекта, чтобы у нас массив поделился на две части: в первой части числа меньше или равные x, а во второй части числа больше или равные x. Далее, точно так же для каждой части мы будем выбирать свои x и делать точно так же. В итоге массив окажется отсортированным. И если части будут примерно одинаково размера, то сложность данного алгоритма будет составлять NlogN. Сколько памяти потребляет быстрая сортировка в худшем случае?

Как предлагал сделать Хоар? Он предлагал взять случайный элемент из массива, завести два указателя: левый и правый. Пока элементы по указателям стоят правильно, мы ничего не делаем. В каком смысле правильно? Если в начале массива стоит число меньше x, то всё хорошо. Мы сдвигаем указатель. Сдвигаем, сдвигаем, и сдвигаем до тех пор, пока не наткнёмся на число, которое стоит неправильно. Неправильное для левой половины - это число больше чем x. После этого мы начинаем искать неправильное число в смысле правой половины, так же сдвигая указатель. Если число больше или равно x, то всё хорошо, иначе нет. И мы swapаем два этих числа. Это и есть метод Хоара. После того, как указатели сойдутся, мы радуемся, потому что у нас стал правильный массив. Но написать это практически невозможно. Если вас просят такое написать на бумажке - вас валят.

Но мы сделаем проще: мы разделим массив на 4 части! Первая часть с числами, которая меньше x, вторая - равными x, третья - больше x и четвёртую мы еще не обработали. Мы всегда будем поддерживать такую штуку. Также нам нужно три указателя: на границу между первой и второй частью, второй и третьей и третьей и четвертой. И приходит к нам очередной элемент (по умолчанию он попадает в четвёртую часть). Если этот элемент больше x, то мы ничего не делаем. Если пришел элемент, равный x, то мы swapаем его с тем элементом, который стоит первым в третьей части. После этого он попадает в группу равных иксов, а здесь всё корректно. Но последний случай самый сложный: когда эле-



мент меньше x. Этот элемент мы сначала запомним, потом первый элемент из третьей части перенесём на его место и таким образом освободим место для первого элемента из второй части. И теперь наш элемент, который мы запомним в самом начале, перекладываем на ту первую позицию из второй части, которую мы освободили. Поэтому у нас сдвигаются все указатели. Опорным элементом следует брать медианный элемент, но нам нужно найти его за O(1). Если брать случайный и нам будет постоянно не везти и будет выпадать последний элемент, сложность алгоритма в худшем случае будет $O(N^2)$. Такой алгоритм будет прекрасно работать на массиве из одинаковых чисел, и будет работать за O(N). Например, в языке C++ quicksort используют в качестве основы, но если он постоянно валится, то используют heapsort. Для каждой области мы запускаем рекурсию. Какая глубина будет у такой рекурсии? В среднем logN, но в худшем случае O(N). И так написано в любом учебнике.

ТАЙНОЕ ЗНАНИЕ: В худшем случае быстрая сортировка потребляет логарифм памяти. Сделать это очень просто: мы каким-то образом порезали массив на части, который нужно посортить, для меньше части мы запустим рекурсивную функцию, а большую продолжим сортировать.

Он очень хорошо работает на частично отсортированных массивах, но не обладает устойчивостью.

Задача о нахождении k-го элемента. Дана задача. Вот мы отсортили массив и посмотрели, кто у него стоит на k-ом месте. Сделать это можно за линейное время. Мы сделаем partition, и допустим наш partition вот такой сразу с тремя частями. И допустим k у нас в третьей части. Где мы должны продолжать искать k-ый элемент? Только в третьей части, потому что в первой и второй части все элементы строго меньше или равны x, и они столько же места и будут занимать. Если мы попали во вторую часть, то вот она: мы уже нашли этого человека. А если в первую часть, то ищем только в ней. Мы просто запускаем partition для оставшейся части. В среднем случае добиваемся сложности $O(N)$, а в худшем случае N^2. Следующий вопрос для экзамена: потребление памяти алгоритма поиска k-ой порядковой статистики за линейное время. O(1) памяти у нас потребляется. Работает за минимум в среднем случае. Какое потребление памяти у быстрой сортировки? В лучшем случае O(1), а в среднем и худшем O(logN).

5.2 Сортировка подсчётом

Мы считаем сколько раз встретился тот или иной элемент а потом его печатаем столько же раз. Увеличиваем счётчик на 1 при встрече с каким-либо элементом. Этот счётчик у каждого элемента свой. Сложность такого алгоритма $O(N + K)$, где N - длина последовательности и K - кол-во возможных значений. Нельзя написать сортировку быстрее, чем за logN, если у вас на руках есть только ответы на вопросы «Да» или «Нет». Эта сортировка выгодна на сортировка при маленьких значениях. Является ли она устойчивой? Эта сортировка является устойчивой, так как два элемента с одинаковыми ключами будут добавлены в том же порядке, в каком просматривались в исходном массиве. Это будет работать практически за линейное время при маленьких K.

5.3 Поразрядная сортировка

Цифровая сортировка — один из алгоритмов сортировки, использующих внутреннюю структуру сортируемых объектов. Предположим, у нас даны трёхзначные числа. Сначала отсортируем их по последней цифре, потом по второй, затем по третьей. Магия! Оно отсортилось. Мы используем сортировку подсчётом, чтобы отсортировать числа в каждом разряде. Сложность такого алгоритма $O((N + K) \times P)$, где P - это количество разрядов. Пример: давайте возьмём стандартное 32-ухбитное число. 4 байта и нарежем по байтам и условно возьмём последовательность длинной 10^9. Получим $(10^9+256)\times 4$. Примерно $(4*10)^9$. Посчитаем

NlogN: равен он $(30 * 10)^9$. Необходимо помнить, что $10^3 \approx 2^{10}$.

5.4 Хеш-таблица

Теперь мы будем использовать эту идею для реализации множества чисел. Как происходит в математическом смысле: каждое число либо входит, либо не входит в множество. Если конечное множество, то будем делать сейчас хеш-таблицу. Предположим, что студент решил собрать коллекцию оценок от 0 до 10. set() использовать нельзя. Что делать? Во-первых, не расстраиваться. Во-вторых, как реализовать множество для таких маленьких чисел? Обычный массив. Массив из 11 элементов. Тебе говорят добавь число 3. Добавляю и ставлю True. Тебе говорят добавь число 7. Добавляю и ставлю True. Тебе говорят посмотри число 8, но в ячейке его нет и стоит False, значит, его нет. Как убрать элемент? Заменить True на False.

Предположим, что нам нужно распределить людей по первой букве фамилии. Это называется хеш-функция от фамилии: то есть вы чему-то большому сопоставляет нечто маленькое. В зависимости от частота фамилии можно по разному сопоставлять им ячейки (в нашем случае) от 0 до 9. Хорошая в данном случае хеш-функция - эта такая функция, которая равномерно распределила людей по ячейкам. Нам надо придумать хеш-функцию, которая переводит числа от 10^9 от 0 до 9, и чтобы она была хорошей. Отсортируем их по последней цифре серии их паспорта: у кого-то в серии куча нулей, а на последнем месте стоит 7. Ставим галочку в ячейку 7. А потом приходит человек, у которого уже везде единицы кроме последней 7. Но у нас уже есть человек с последней цифрой 7... Поэтому такая идея с галочками просто так не работает, но работает идея со столами. То есть мы будем числа записывать в ячейки в явном виде. Я хочу добавить число 17, я записываю его в ячейку 7. Я добавил число 23, я записываю его в ячейку 3. Я добавил число 37. Что я делаю? Я буду хранить списочек таких чисел внутри ячейки, у которых последняя цифра - это 7. Это один из возможных вариантов реализации хеш-таблицы. Есть еще другие реализации: это называется открытая и закрытая адресация. Но раньше экономили от бедности, поэтому способ выше лучше. Когда у нас просят, есть ли у нас число 47, то должны будем определить значение хеш-функции (семёрка) и линейным поиском поискать в седьмой ячейке число 47 среди всех чисел с такой хеш-функцией.

Операция добавления: O(1). Операция поиска: O(N//K). Операция удаления: O(1) в среднем. Напомним, N - количество различных элементов,

а K - длина таблицы. От значений ничего не зависит. Мы реализовали две основные операции множества: добавь, проверь и удали. Но еще есть еще одна приятная операция: вывести все содержимое множества. То есть мы пробежимся по всем значениям множества и выведем всё содержимое списков в каждой ячейке. Упорядоченности не будет. В чём проблема такого подхода? Если мы возьмём большое K, то у нас очень много памяти будет занимать даже маленькое множество. Если мы возьмем маленькое K, то большие множества, в которых много элементов, будут работать медленно. Как только мы достигли какого-то коэффициента заполнения (желательно 0,7), то мы увеличиваем хеш-таблицу вдвое и перестраиваем ее, перекладывая элементы. Именно так устроено множество set() в языке Python.

6 Два указателя. Сортировка событий

6.1 Два указателя

Пусть у нас есть последовательность чисел, числа неотрицательные. И нам необходимо найти такой подотрезок, сумма элементов которых задана числом S. Дан массив 2 1 3 4. Нужно такой отрезок определить. Как решить такую задачу? Вы можете воспользоваться тем, что можете запоминать суммы префиксные. То есть у вас были числа какие-то и вы смотрите на сумму префикса. Вот у вас числа 2 3 6 10 и кладёте их в set. Например, s = 7 над элементами 3, 4 и -.

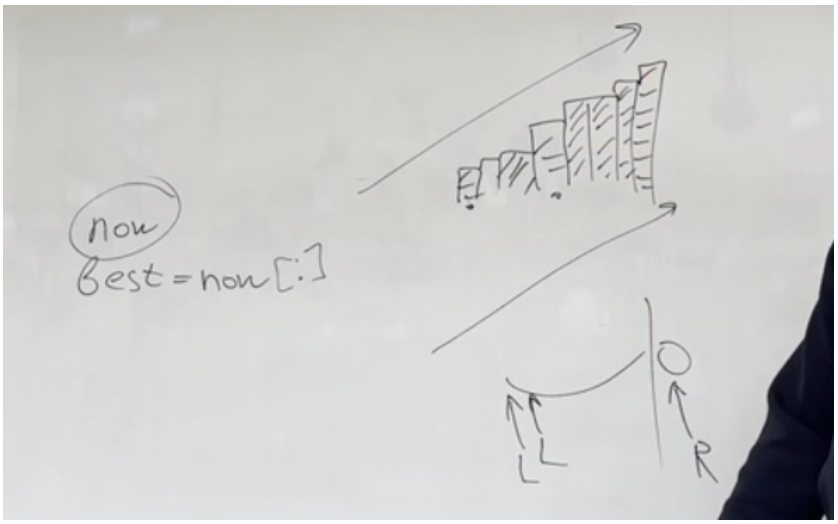
2	1	3	4	-	-
2	3	6	10	-	-

10 - 7 = 3. Если три встречалось раньше, то это значит, что у нас существует отрезок сумма элементов которых равна 7. Но это не спортивно. Самый глупой способ: мы будем перебирать всевозможные левые и правые границы отрезка за N^2. И потом еще раз вложенным циклом пробегаться по отрезку от L до R и считать сумму на этом отрезке. Такой решение занимает N^3. Но мы оптимизируем задачу до N^2. Мы перебираем L, а R мы будем постепенно двигать, правую границу, и прибавлять каждый раз то число, которое у нас новое появилось. Тогда мы будем поддерживать актуальную сумму от L до R, пересчитывать ее за O(1) от предыдущей суммы. Но это всё еще медленно. Смотрите, как только, сумма между L и R больше заданного числа s, мы можем закончить работу. Но в худшем случае это нам не даст никакого ускорения. Мы сейчас ее оптимизируем. Допустим, R где-то остановилось, в какой-то момент. И мы впервые попали в ситуацию, для данного L, где сумма больше чем s. Сейчас мы будем переходить к следующему L. Идея в чём: то что мы эту сумму можем использовать. Мы понимаем, что нет смысла перебирать R меньше, чем сейчас, потому что сумма элементов будет меньше, чем s. Что это значит? Это значит, что мы можем передвинуть L на 1 и считать сумму между двумя указателями. Это и есть метод двух указателей. Как он пишется?

```
R = 1
for L = 1, N - 1:
    while R < N and good(R):
        ...
        R += 1
```

Сложность такого алгоритма $O(N)$. Общая схема почти всегда такая, как представлена в коде. Но надо отдельно обрабатывать случай, когда указатель наткнулся на конец массив - это тоже нужно учитывать. Когда вы будете тестировать своё решение, необходимо так же учесть этот случай.

Задача про игроков. Условие: надо собрать команду игроков так, чтобы двое любых членов были сильнее самого сильного игрока (неравенство треугольника). Отсортим наше множество игроков: всегда, когда не понимаете что делать, сортите элементы. Возможно, это вам поможет. Теперь нам нужно найти некий отрезок в отсорченной последовательности, а это в свою очередь будет сплоченной командой, которая нам необходима.



Наша задача найти все такие отрезки. R будет двигаться до такого игрока, которого мы не можем взять. Значит, R будет показывать за границу сплочённой команды. По этой схеме действуете и перебираете вложенным циклом. Здесь прекрасно метод двух указателей: мы смотрим на элементы L и L + 1 и сравниваем его с R. И так дальше сдвигая указатели. Но есть несколько нюансов. Во-первых, тот самый случай, когда мы прижаты к правому краю. Как наша реализация будет себя вести? Напомним, наш вложенный while выглядит как while с двумя условиями. Теперь давайте подумаем, может ли самая сильная команда состоять из одного человека? Да, потому что к самому сильному человеку можно добавить кого угодно и всё равно будет сплаченная команда. За одним

исключением: когда у нас в массиве всего один элемент. Теперь усложним задачу: назовём номера игроков, которые образуют сплоченную команду. То есть находя нужных игроков, мы будем выводить их и дальше пробегаться по ответу.

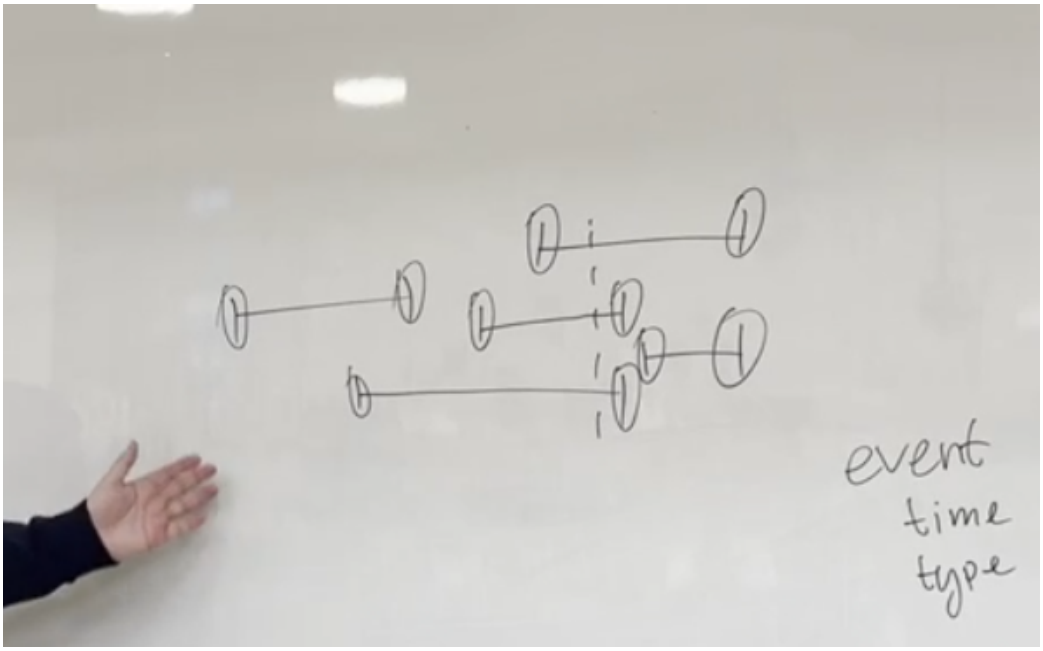
Задача об аминокислотах. Условие: Дана ACTG и строка AACATTAGCA. И нам нужно найти подстроку, которая содержит в себе все возможные четыре аминокислоты. То есть самая короткая строка, где встречаются ACTG. Смотрите, я ставлю левый указатель на первый элемент, а второй указатель на тот элемент, на момент которого встречаются все необходимые нам буквы. Для этого нужно поддерживать словарь. А когда мы двигаем левый указатель, то вычитаем единичку. Двигаем, двигаем, понимаем, что теперь уже не все элементы есть в подстроке, поэтому мы начинаем двигать правый указатель. Двигаем, и двигаем до нахождения ответа. Сложность такого алгоритма O(NA).

Задача о длине подстроки. Найти самую длинную подстроку, в которой каждая буква встречается не более k раз. В этой задаче не нужно проверять это внутри while. Мы двигаем правый указатель до тех пор пока у нас всё хорошо. Теперь мы двигаем левый указатель на один. И мы будем двигать его, пока не наткнёмся на элемент, на который наткнулись до этого указателем R, то есть теперь мы L двигаем до тех пор, пока не станет хорошо, уменьшая количество вхождений на 1, через которые L прошёл. Потом снова начинаем двигать R, пока снова всё не станет плохо. Сложность этого алгоритма O(N).

Задача о двух парнях. На проспекте есть несколько памятников и девушка решила устроить свидания с двумя парнями сразу. Свидание назначается у памятников, поэтому считается, что они друг друга не видят, если стоят на расстоянии d и больше. Давайте перебирать все памятники в качестве левого указателя, а правый указатель будет показывать на первый памятник, который находится на расстоянии d или больше. Что не так? Стоит помнить про случай, когда R может уйти за последний памятник.

6.2 Сортировка событий

Часто оказывается, что у нас есть какие-то отрезки, как на картинке.



Дана задача: сказать, сколько пользователей было одновременно на сайте. Ответ: 3. Как ее решать? Что-то интересное происходит только тогда, когда человек приходит и уходит с сайта. Между этими моментами ничего не происходит. А значит, мы можем смотреть только в эти моменты. Мы возьмём эти моменты времени и породим два события: человек пришел и человек ушёл. Мы складываем их в один массив и сортим. Отсортированные по возрастанию моменты времени, когда происходит что-то интересное. Что у нас должно быть в событии? Мы должны знать его тип: пришел или ушел. В зависимости от задачи у нас могут попросить что-то дополнительное. Мы их посортили и теперь смотрим: если человек пришел, то делаем +1 к счётчику, если человек ушёл, то -1. И максимальное значение, которое достигал этот счётчик, это и есть ответ.

А если кто-то одновременно пришел и ушёл? Как считать? В хорошем смысле сначала надо добавить, а потом вычесть. А в плохом смысле, то есть поменьше, сначала человек уходит, а потом приходит. На самом деле это будет ясно из условия. По такому принципу решается огромное кол-во задач. Например, нужно показать все промежутки времени, когда сайт испытывал максимальную нагрузку. То есть вам надо посчитать суммарное количество людей, которое было на сайте. Есть один простой способ. Вот у вас событие: максимальное кол-во посетителей. Вот идея: когда у вас максимум перестал быть максимумом, то есть ушел один

человек. Далее вы просто прибавляете разницу во времени между текущим моментом времени и прошлым, когда вы только-только достигли максимального значения.

Задача о парковке. На парковке есть секции. Каждое событие описывается следующим образом: время приезда машины, время отъезда машины и места, которые она заняла. Нам нужно выяснить, была ли парковка хотя бы раз занята полностью? По очевидным причинам машины друг на друга не становятся, поэтому мы можем таскать с собой кол-во занятых мест. Тип события можно кодировать через значок плюс или минус. Если счётчик стал равен количеству мест на парковке, мы говорим, что да, был случай, когда парковка забита полностью.

Усложним задачу: нам нужно не только выяснить, была ли забита парковка, но и понять, каким минимальным числом машин она была забита. После каждого события мы проверяем такую ситуацию, не занята ли у нас вся парковка. Если занята, то мы обновляем минимум. И в конце проверяем, была ли такая ситуация, и выводим ответ.

Усложним еще сильнее: сказать номера машин, которыми она была занята. К каждой машине мы прибавляем еще один параметр: номер. То есть нам нужно сказать номера машин, которые своим минимальным количеством заняли всю парковку. Ответ: берем set() номеров и обновляем его каждый раз когда улучшили ответ. Сложность такая алгоритма будет O(NlogN), решая через два указателя.

7 Динамическое программирование. Классические задачи

Что такое динамическое программирование? Скорее всего, вы знакомы с математической индукцией, а значит, можете найти много общего. Потому что это способ посчитать оптимальные решения задачи или колво решений задачи через решение меньших задач. Давайте рассмотрим на примерах.

Задача о лестнице. У нас есть лестница и N количество ступеней. Нам нужно посчитать количество способов подняться на эти N ступеней. При том, что можно шагать на следующую ступень или через одну. Решать мы это будем по аналогии с базой индукции: на первую ступень с нулевой ступени у нас есть один способ добраться. А дальше нужен переход. То есть нам нужно выразить нашу задачу через задачи меньшего размера. Давайте мы обозначим это как dp[i]. То есть способов попасть в эту ступеньку у нас равна сумме способов попасть на прошлую ступеньку и на позапрошлую. В коде это выглядит так:

```
dp[i] = dp[i - 1] + dp[i - 2]
```

Но необходима база, потому что по этой формуле мы не сможем посчитать кол-во для первой ступеньки. И в таком случае у нас два варианта: либо вы используете еще какие-то формулы и рассматриваете общий случай, либо вы придумываете виртуальные элементы и придаёте им какие-то значения. Первый способ - это посчитать ко-во для первого, а дальше наша формула будет работать. Второй способ - мы заводим какую виртуальную -1-ую ступеньку и поскольку она будет виртуальная, ее значение будет 0. То есть у вас 0 способов подняться на ненастоящую ступеньку. Эта формула напоминает числа Фибоначчи, и сейчас мы научимся их считать за логарифм. Предположим, у нас есть матрица слева и нам нужно придумать, на какую матрицу ее умножить, чтобы получить матрицу справа:

$$\begin{pmatrix} f_i \\ f_{i+1} \end{pmatrix} = \begin{pmatrix} f_{i+1} = f_{i+1} \\ f_{i+2} = f_{i+1} + f_i \end{pmatrix}$$

Это матрица: $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$

То есть $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} f_i \\ f_{i+1} \end{pmatrix} = \begin{pmatrix} f_{i+1} \\ f_{i+1} + f_i \end{pmatrix} = \begin{pmatrix} f_{i+1} \\ f_{i+2} \end{pmatrix}$ То есть, когда мы умножаем матрицу из нуля и единиц на числа Фибоначчи, мы получаем следующие числа Фибоначчи. Если мы возьмём k-ую степень этой матрицы и умножим на числа Фибоначчи, то получим на k чисел Фибоначчи вперед.

А как быстро матрицу возвести в степень? Этот алгоритм мы рассмотрим позже, но в чём он заключается? Мы берем следующие соотношения: если вдруг степень оказалась чётной, то вы можете возвести число в квадрат и уменьшить степень вдвое. Отсюда берется логарифм. А если она окажется нечётной, то вы можете один раз сделать умножение и дальше работать уже с чётным числом. И таким образом на каждом шаге вашего алгоритма показатель степени будет уменьшаться вдвое, что даст логарифмическую сложность возведения в степень. Пишется это рекурсивно. Но есть проблема: числа Фибоначчи растут очень быстро. И операции по типу сложить или умножить выполняются, конечно, за O(1), но пока числа маленькие. Как только числа становятся большими, они начинают выполняться долго.

Рассмотрим следующую вещь: теперь нам можно подниматься на одну, две или три ступеньки. Количество способов будет считаться так:

```
dp[i] = dp[i - 1] + dp[i - 2] + dp[i - 3]
```

И таких случаях, когда подсчёт базы не такой простой, иногда оказывается эффективно ввести виртуальную ступеньку. И если в этой задаче мы можем сделать целых 2 фиктивные ступеньки, то можно вручную посчитать для одной виртуальной и самой первой, чтобы затем опираться на них. Универсального решения нет. Матрица для таких чисел будет выглядеть так:

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}$$

Это первый тип задач из динамического программирования: когда нам нужно посчитать количество решений.

Теперь представим, что на некоторые ступеньки нельзя наступать. Про-

сто при вычислении в такие ступеньки вы просто записываете число 0. И считаете по формуле, но тут уже посложнее будет с матрицей.

Задача о платной лестнице. Теперь уже, поднимаясь по ступенькам, у нас за это будут брать деньги! Мы можем либо на одну либо на две вверх подняться. За то что мы поднимаемся на каждую ступеньку мы должны заплатить a_i-ое кол-во рублей. Нам нужно найти минимальное количество денег, которое мы можем потратить. На первой ступеньке мы стоим за ноль рублей, да и на первой виртуальной тоже. Как будет выглядеть самый дешевый способ добраться до i-ой ступеньки, будучи на ней стоять?

```
dp[i] = min(dp[i - 1], dp[i - 2]) + a[i]
```

Задача о складе. Есть склад, крышу склада держат столбы и сторож хочет сдать эти столбы в металлолом. Но незадача: крыша склада обрушивается, если убрать два идущих подряд столба. Каждый столб весит a_i. Нам нужна следующая формула:

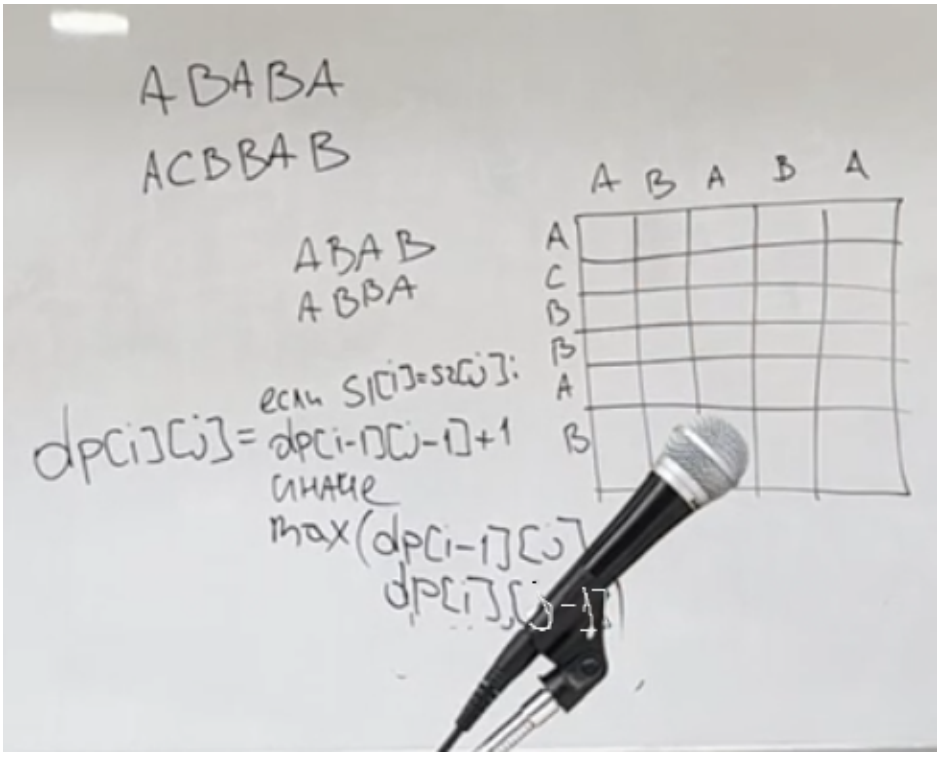
```
dp[i] = max(dp[i - 1], dp[i - 2] + a[i-1])
```

Здесь нулевой столб будет виртуальным, а также с первого столба мы зарабатываем 0, то есть мы считаем начиная со второго. Но по условию задачи может оказаться так, что мы искали до N-го столба, но продать его могло было бы быть выгодно. Что же делать? Посчитать dp[N+1], где N+1 - виртуальный столб. Теперь нам нужно сказать сторожу, какие именно столбы он должен спилить, чтобы сдать в металлолом. Это называется восстановление сертификата. Первый способ заключается в том, чтобы мы дополнительно ничего не хранили. То есть мы развернули формулу в обратную сторону и так же ее используем, чтобы понять, пилить или не пилить. В одном случае мы ничего не добавляем в ответ, в другом - добавляем $N - 1$. Всегда у нас в динамике ответ восстанавливается задом наперед. В дополнение к этому мы заводим массив, где записываем значения, которые запомнили.

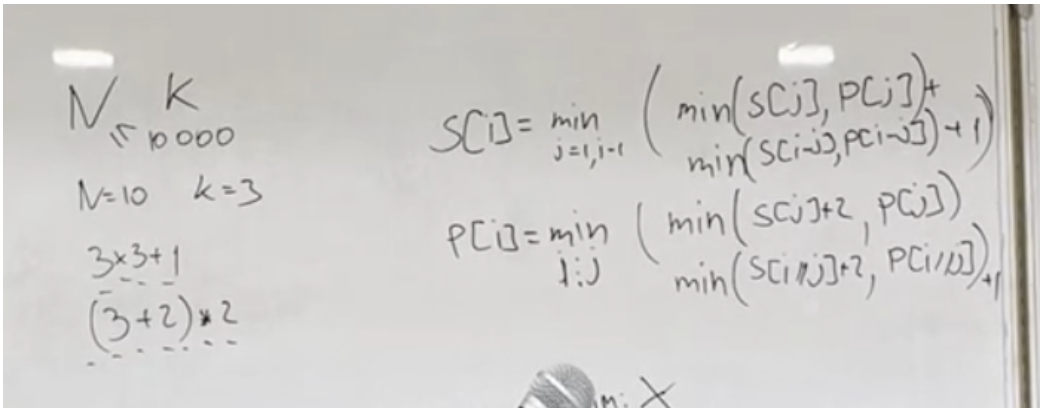
Задача о наибольшей возрастающей подпоследовательности. У вас есть некоторая последовательность чисел. И необходимо найти длину самой большой подстроки с возрастающей последовательностью.
Придаём смысл dp[i]: смотрим, к каким элементам до этого можно приписать текущий элемент и смотрим наилучший для нас вариант. Такой алгоритм можно написать за NlogN, но я Вам его не расскажу, потому

что он очень специфичный. Теперь нам нужно восстановить путь: мы смотрим на элемент, с которым совпадает число длины максимальной последовательности и ищем линейным поиском элемент, который совпадает с числом длины максимальной последовательности меньшим на единицу, и так далее. Или мы можем просто запоминать индекс элемента, который мы использовали для подсчёта. Времени это занимает N^2.

Задача о наибольшей общей подпоследовательности. У нас есть две строки ABABA и ACBBAB. Какая самая длинная общая подпоследовательность? Эта задача имеет практическое применение: оно помогает исправлять опечатки в словах. Создаём двумерный массив, и строчки подписываем буквами из второй строки, а столбцы - буквами из первой строки. На картинке изображена формула:



Задача о записи числа. Даны числа N и K. Вам необходимо записать число N в виде арифметического выражения используя плюс, умножить и скобки и числа, не превосходящие K, с наименьшим количеством символов в записи. На картинке показан пример и формула для подсчёта. Заведем два массива: оптимальное представление числа в виде суммы

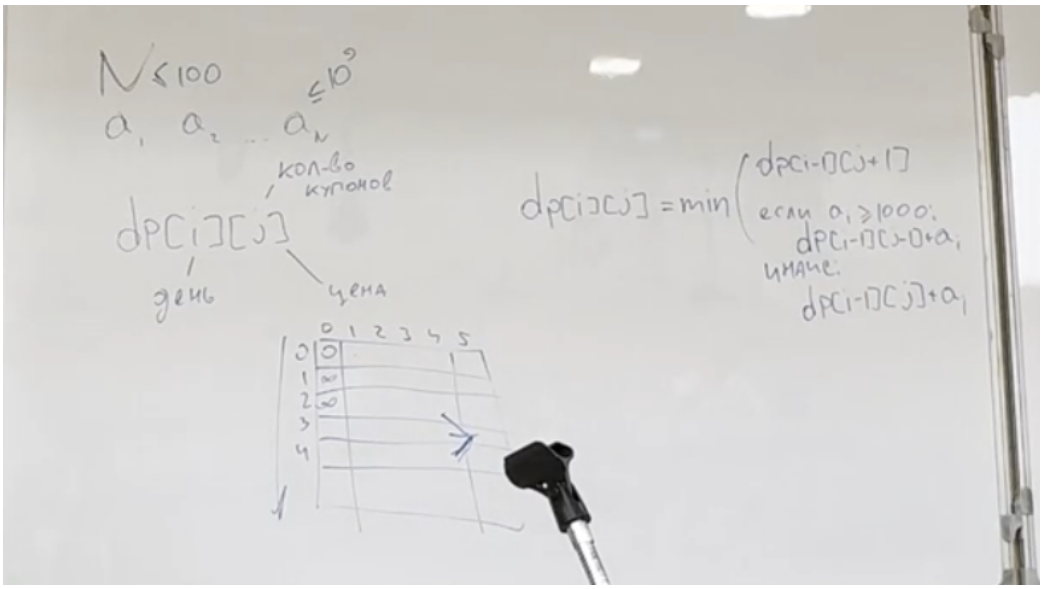


и в виде произведения. Их база будет равна количеству знаков в числе K. Где искать ответ? Смотрим, кто лучше: $s[n]$ или $p[n]$. Теперь восстанавливаем ответ. Время работы этого алгоритма N^2. Памяти занимает столько же.

8 Двумерная динамика. Динамика по подстрокам

Сегодня мы разберем задачи, параметры в которых имеют разную природу. Что нам нужно помнить? Какой смысл мы придаём массиву, в котором храним какую-то информацию, формула пересчёта, начальные значения, в каком порядке считать и где ищем ответ.

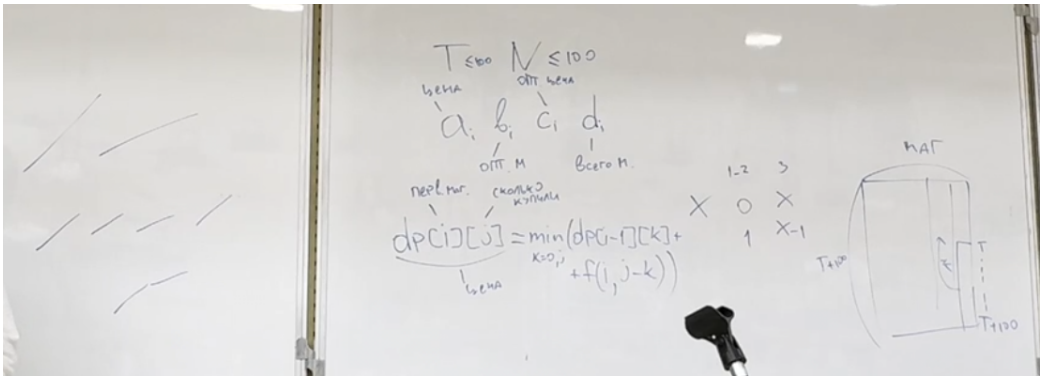
Задача про суп в столовой. У нас есть N дней. И каждый из N дней у нас известна цена обеда в этот день, она различна для каждого из дней. При чём есть мы можем либо за деньги, либо за купоны, а купон дают, если мы поели обед, который стоит больше 1000 рублей. Задача: есть каждый день как можно дешевле. Нам необходимо оптимизировать два параметра: цену минимализировать и количество купонов максимизировать. Сделаем мы это вводом дополнительного параметра.



В таких задачах, выбирая такой параметр, обычно выбирается меньшее из чисел. Так же в условии может быть сказано, что нужно выбрать такой способ, чтобы количество купонов было наибольшим. Где с таким условием искать ответ? В последнем столбце минимальное число, а если таких несколько, то выводим последнее. Но пробегать по столбцам дольше, чем по строкам, поэтому, если вы гонитесь за производительностью, нужно [i] и [j] поменять местами.

Как восстанавливать ответ? То есть выводить, в какие дни мы ели за купоны? Либо мы для каждой ячейки создаём таблицу такого же размера, где дополнительно храним тратить мы купон или нет. В зависимости от этого мы всегда при восстановлении ответа будем переходить в предыдущий столбец (если работаем со столбцами), а если потратили купон, то в предыдущую строку и предыдущий столбец, если не потратили, то варианты: или предыдущий столбец и предыдущая строка, или предыдущий столбец и следующая строка в зависимости от цены обеда.

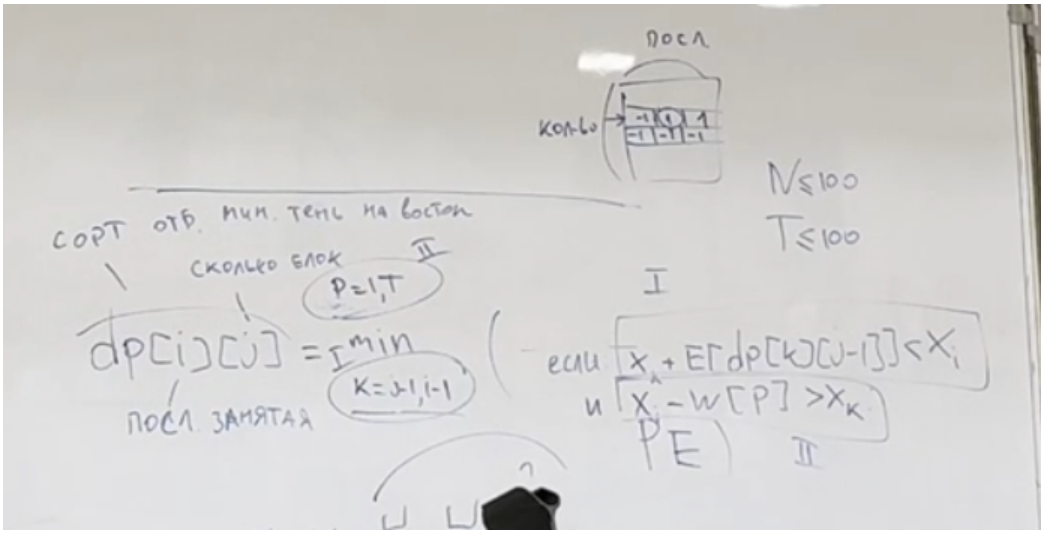
Задача о ткани. Нам известно следующее: цена одного метра ткани, с какого количества метров начинается оптовая цена, оптовая цена и сколько метров ткани продаётся в магазине. Наша задача купить во всех этих магазинах T или больше метров ткани по минимальной цене. Если мы можем купить всё в одном магазине, мы это и делаем. Нам понадобиться вспомогательная функция, которая но номеру магазина будет сообщать цену. Чтобы восстановить ответ, нам нужно запоминать, сколь-



ко метров ткани мы купили в последнем магазине, то есть при каком k достигался минимум.

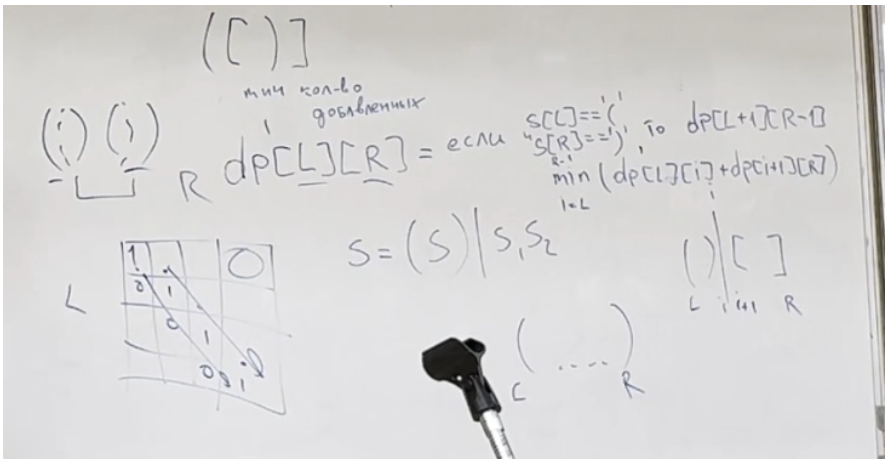
Задача про клумбы. У нас есть аллея, на ней стоят клумбы (у них есть некоторые координаты). Также у нас $N \leq 100$ клумб и $T \leq 100$ сортов ёлок. У разных сортов разные характеристики. Ботаники выяснили, что ёлка растёт плохо, если на неё падает тень. Ваша задача посадить максимальное количество сортов ёлок так, чтобы они чувствовали себя хорошо. Первой мы сажаем ёлку, которая отбрасывает минимальную тень на восток. Но также, может быть случай, что даже если мы и можем посадить ёлку, мы должны ее пропустить, потому что тень на эту клумбу падает с двух сторон и этого не получается избежать. Это говорит

о том, что жадный алгоритм не работает. Чтобы в последствии восстановить ответ, вместо минимальной тени на восток нам нужно хранить посаженный сорт. Ну и таким образом мы определим тень и сорт. Циклы в данной задаче не вложенные, а последовательные! Ответ на задачу мы ищем в табличке: последнее занятое и количество. В ней нам достаточно найти максимальный номер строки, в которой есть хоть что-нибудь отличного от признака невозможности.



8.1 Динамика по двум строкам

Дана скобочная последовательность. Необходимо определить мини-



мальное количество скобок, при добавлении которых она станет правильной. База находится под главной диагональю в табличке, то есть

мы обращаемся к ней, когда левая граница становится больше правой. Ответ будет хранится в [1][R]. Большинство задач на динамику по подстрокам решается именно таким образом.

8.2 Рекурсия с оптимизацией

Чтобы не проходится по диагонали в табличке, можно воспользоваться следующей функцией:

```
f(l, r):
    if dp[l][r] == -1:
        for i = L, R - 1:
            dp[l][r] = min(f(l, i) + f(i+1, r)
        if ... (на случай совпадения скобок)
    return dp[l][r]
```

9 Динамическое программирование. Задача о рюкзаке. Жадные алгоритмы

9.1 Жадные алгоритмы

Научимся доказывать, работает ли жадный алгоритм или нет на примерах.

Задача 1. У нас есть N людей. Для каждого человека известно расстояние в километрах, которое он должен проехать. А так же у нас есть N машин такси и для каждой машины известная стоимость за 1 километр. Наша задача: отправить как можно больше людей по одному в такси за минимальную стоимость. Общая идея: мы отсортируем людей по возрастанию, а такси наоборот по убыванию. Утверждается, что человек, которому ехать меньше всего, должен ехать на самом дорогом такси, а человек, которому ехать дальше всего, - на самом дешевом. То есть чем дальше человек едет, тем дешевле такси он должен использовать.

А почему оно работает? Допустим, у нас существует какое-то другое правильное решение. То есть у нас найдётся такая пара i, j, что расстояние i-го человека меньше расстояния j-го. Чтобы у нас возникло противоречие, мы скажем, что стоимость такси тоже меньше. То есть нарушение того, что мы сказали сверху. И есть некоторая сумма, которую мы заплатили за поездку всех людей в такси. Что мы можем сделать? А что если мы поменяем их местами. Теперь i-ый человек будет ехать на j-ом такси, а j-ый человек - на i-ом такси. Чтобы прийти к противоречию, то есть доказать, что такой пары нет, нам нужно показать, что изменившаяся стоимость $dist_i \times cost_i + dist_j \times cost_j = s$ строго меньше, чем $dist_i \times cost_j + dist_j \times cost_i = s'$. Но в лоб это доказать сложно.

Пусть $dist_j = dict_i + x$, $cost_j = cost_i + y$, где x, y > 0. Подставляем:
1) $dist_i \times cost_i + (dict_i + x) \times (cost_i + y)$
2) $dist_i \times (cost_i + y) + (dict_i + x) \times cost_i = dist_i \times cost_i + dist_i \times y + dist_i \times cost_i + x \times cost_i$
После сокращения слагаемых с первым выражением, получаем: xy и 0. Следовательно, $s > s'$.
То есть доказывая, мы предполагаем, что существует какое-то лучшее решение, а затем показываем это.

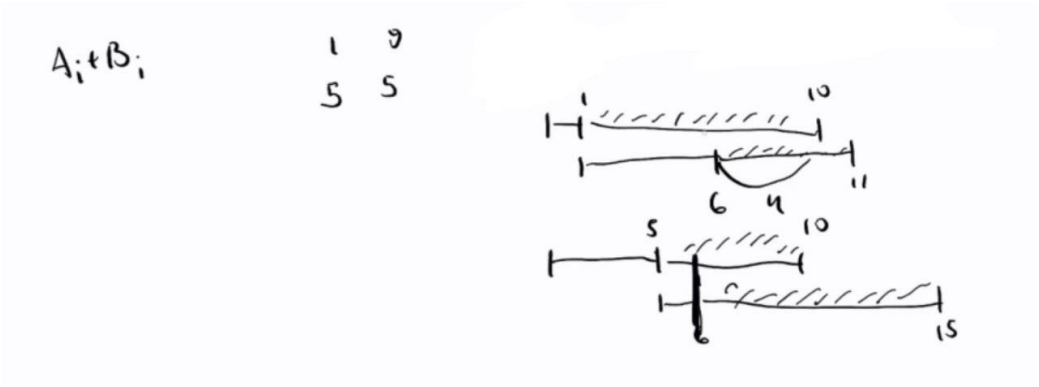
Задача 2. У нас есть некий сосуд с жидкостями A и B. Нам надо разделить их пластинками. Каждая пластинка обладает одной из характеристик: A_i и B_i - время, за которое жидкость A разъедает эту пластинку, и время, за которое жидкость B разъедает эту пластинку соответственно. Если есть время, то мы можем придумать такую штуку, как скорость разъедания пластинки. И если у нас одна пластинка, то она суммарно разъедается двумя жидкостями. Нам нужно расставить пластинки в таком порядке, чтобы жидкости как можно дольше не встретились между собой. Если у пластинки большое A, то мы вроде как должны ставить ее слева, а если у нее большое B, то должны ставить ее справа.

Если мы посортим по убыванию A_i/B_i, то получим последовательность, которая будет дольше всего растворяться. Если значения одинаковые, то их расположение относительно друг друга не играет роли. Почему это работает? Можно себе представить, что пластинки сделаны из одного материала, то просто разной толщины. Проверим: сперва расставим A_i и B_i, затем B_i и A_i. Они между собой равны, поэтому наше утверждение, что A_i/B_i, верно, потому что с высокой доли вероятности, если был пройден этот тест, то и остальные тоже пройдут.

Задача 3. У нас есть Белоснежка, она хочет уложить гномов спать. Чтобы i-ый гном улёгся спать, он должен на протяжении A_i кол-ва времени слушать колыбельную и потом ещё он спит в течении B_i кол-ва времени. Причём каждому гному нужно отдельно петь колыбельную. И после того, как первый гном уснул, мы можем начать петь колыбельную j-тому гному. Нам нужно понять, существует ли такой порядок распевания колыбельных, что мы сможем уложить всех гномов спать так, чтобы они все спали одновременно, и сказать порядок.

Идея такая: гному, который дольше засыпает и спит, поём колыбельную в первую очередь. То есть сортируем гномиков по убыванию $A_i + B_i$.
В таком случае, для двух гномов с одинаковыми значениями $A_i + B_i$, показатель того, когда засыпает последний и просыпается первый не меняются, а значит, их расположение не имеет значения.

Вот такая явная может быть для проверки жадного алгоритма на некорректность. Это не является доказательством корректности алгоритма, но с большой вероятностью, если тест пройден, то алгоритм корректен. Как доказать его? Опять-таки ищем такую пару, меняем их и приходим к вы-



воду, что ухудшиться не могло, а либо улучшилось, либо осталось также, как мы делали это в начале.

9.2 Задача о рюкзаке

Человек собирается в поход и кладёт в сумку наиболее ценные предметы. Например, в сумке лежит песок и алмазы. W_i - количество, а $_i$ - ценность. И вот с таким рюкзаком он решил пойти в поход. Почему песок и маленькие алмазы? Потому что я привожу пример, когда мы можете взять любую часть предмета. То есть у нас есть рюкзак с грузоподъёмностью V и нужно набрать максимальную суммарную ценность в этот рюкзак. И в моём случае я могу делить предметы. Я часть алмазов отсыплю, а часть оставлю дома: может быть, не пригодятся. Какие предметы класть в рюкзак? Очевидно, с самой большой ценностью. Набрали одного - переходим к другому. И так по убыванию их стоимости сортируем. Теперь сделаем переход.

Пусть у нас будет не песок, а какие-то предметы. И у каждого предмета есть вес W_i и стоимость C_i. И мы хотим взять не более, чем V килограмм предметов, чтобы их суммарная стоимость была наибольшей из возможных. По аналогии с задачей о делимых предметах, мы можем посчитать удельную стоимость килограмма предмета. То есть для каждого предмета придумать такую характеристику, что W_i/C_i. Идея: посортить предметы по этой характеристике по убыванию. Но это неверное решение, потому что найдётся контрпример. Эта задача класса NP, поэтому в общем случае она не решается (точнее решается, но за экспоненциальная время). Поэтому немного упростим задачу: пусть предметы пока не обладают стоимостью, а обладают только весом. И наша цель: просто унести как можно больше. А также вес предмета обязательно должен

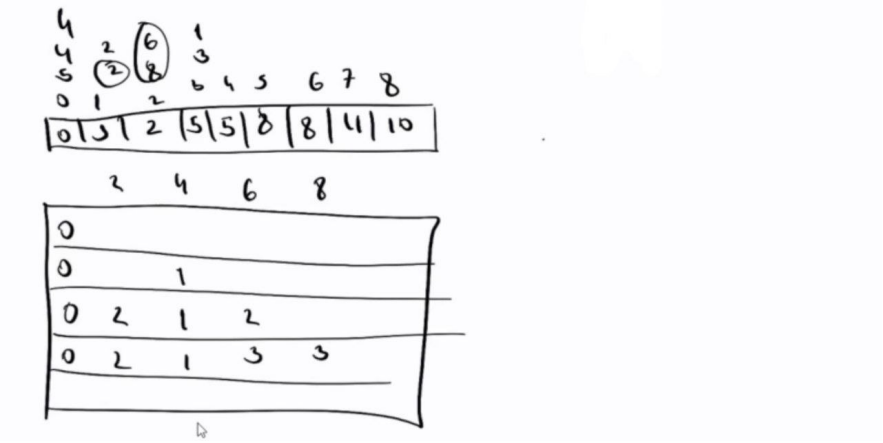
быть целым числом. Как мы будем рассуждать? Я создам массив длинной V + 1 = 11 и потихонечку начну обрабатывать предметы. Как я буду их обрабатывать? Предположим, у нас есть предметы весом 6, 4 и 4. В начале, пока я не обработал ни одного предмета, я умею собирать вес 0. То есть, если я умею собирать вес 0, то я также могу собрать вес 4, а с помощью предмета, который веси 6, я научусь набирать вес 10. А еще я могу набрать вес 6. То есть ставлю галочки в массиве во всех местах, которые находятся правее какой-то галочки на вес предмета. Давайте я приложу оставшийся предмет 4. К десяти я его приложить не могу, потому максимальный вес у нас 11. Смотрите, какая у нас может быть опасность: у нас каждый предмет в одном экземпляре, поэтому для меня очень важно не приложить предмет к самому себе. Самый простой способ: это идти по галочкам справа налево и ставить новый галочки: они на каждом шаге точно будут учтены. Напомню, что значат галочки: они означают, что мы умеем набирать какой-либо определённый вес. Если вес не превосходит вес рюкзака, то помечаем. То есть в рюкзаке с грузоподъёмностью 11 и предметами весом 4, 6, 4 мы можем унести вещей максимум на 10 килограмм. Если мы используем полный перебор, то у нас получается, что мы либо берем предмет в рюкзак, либо не берем. И в каких-то узлах в конце дерева перебора, у нас будет получаться суммарные веса. Но, чтобы это работало не за 2^N, где N - количество предметов, мы применили такую оптимизацию и что получается: если мы какой вес умели набирать, то второй раз рассматривать не будем такой вариант. И за счёт этого сложность перебора становится O(VN), где V - объём рюкзака (обязательно целочисленный) и N - количество предметов. Но это работает только для целых чисел, потому что если числа будут вещественными, то таких повторов может и не быть.

Изменим задачу и скажем, что теперь мне нужно сказать номера этих предметов. Теперь предмет номер 1 имеет вес 4, предмет номер два имеет вес 6 и предмет номер 3 имеет вес 4. Я хочу найти максимальную вес, который я могу положить в рюкзак, и сказать номера предметов, который я в него положил. Как я могу это сделать? Сначала я помечаю, что вес 0 можно набрать неким предметом 0, а всё остальное помечают таким знаком, чтобы было видно, что я не умею набирать такой вес. Например, -1. Теперь я делаю ровно те действия, который делал до этого: я иду и пытаюсь приложить очередной предмет, например, весом 4, ко всем наборам, котоыре у меня были. Но теперь вместо галочки я ставлю номер предмета. Дальше у нас предмет с весом 6: прикладываю к предмету с весом 4 и и ставлю на место 10 номер добавленного предмета, то есть 2. Как восстановить ответ? Допустим была задача набрать максимальный

вес и мне нужно сказать какие предметы я набираю. Я говорю, что беру предмет номер 2 (предмет номер 2 имеет вес 6), а значит, 10 - 6 = 4, и теперь я продолжаю с четвёрки. Там записан предмет номер 1, который имел вес 4, а значит 4 - 4 = 0, а там уже лежит несуществующий предмет.

Вернёмся к изначальной задаче, то есть введем значение стоимости. Что мы будем делать? Мы в этом массиве, где мы ставили галочки, будем ставить не галочки, а суммарную стоимость. Пусть у нас есть всего 4 предмета: 1) весом 4 и стоимостью 5; 2) весом 2 и стоимостью 2; 3) весом 6 и стоимостью 8; 4) весом 1 и стоимостью 3.

Что будет происходить? Пока мы не начали обрабатывать предметы, мы умеем собирать 0 кг со стоимостью 0. Теперь мы учимся собирать рюкзак весом 4 и стоимостью 5. Потом мы встретили рюкзак весом 2 и стоимостью 2: теперь мы умеем собираться рюкзак весом 6 и стоимостью 7 и кроме этого я учусь набирать набор весом 2 и стоимостью 2. Дальше я встречаю предмет весом 6 и стоимостью 8. Чтобы рюкзак не лопнул, я прибавляю этот предмет ко второму набору, и теперь я умею собирать набор весом 8 и стоимостью 10. И теперь я создаю третий набор: шестую ячейку я обновляю и теперь она стала еще лучше, потому что мы туда записали 8, ведь она уже лучше 7! Теперь переходим в последнему предмету - четвёртому. Я могу приложить его к шестёрке и тогда я научусь собирать набор весом 7 и стоимостью 11. Могу приложить его к четвёрке и получу набор весом 5 и стоимостью 8. Могу приложить к двойке и получить набор весом 3 и стоимостью 5. Могу приложить к нулю и получить набор весом 1 и стоимостью 3. А чтобы восстановить ответ, вам придётся хранить много-много строчичек, чтобы там хранить действия.



10 Хеши для строк

10.1 Сравнение двух полиномов

Полином - это многочлен. Полином n-ой степени выглядит так: $a_0x^n + a_1x^{n-1} + ... + a_{n-1}x + a_n$. Если вдруг у нас несколько полиномов, например, такой же с коэффициентом b той же степени, и нам понадобилось сравнить эти два полинома на равенство между собой, то у вас есть два пути: первый - если мы будем сравнивать каждый коэффициент, второй - более изощренный путь, который нам поможет. Мы будем сравнивать кусочки полиномов между собой. Например, возьмём x_0 и посчитаем значение полинома в точке x_0 в обоих полиномах. Если они совпали, то с высокой вероятность мы можем говорить, что они равны между собой. Вроде мы ничего не выигрываем с этого, но если бы у нас было много полиномов и нам нужны было бы разбить их на классы эквивалентности. ТО есть вы могли посчитать для каждого полинома значение в точке x_0 и разложить по этим элементам словарь, где значением выступает сам полином. Какую задачу мы хотим научиться решить? Мы хотим научится сравнивать между собой кусочки полинома. Различные.

Возьмём следующий полином: $a_0x^n + a_1x^{n-1} + a_2x^{n-2} + a_3x^{n-3} + ... + a_{n-3}x^3 + a_{n-2}x^2 + a_{n-1}x + a_n$. Но не получилось ли у меня так, что коэффициенты при $a_1x^{n-1} + a_2x^{n-2} + a_3x^{n-3}$ и $a_{n-3}x^3 + a_{n-2}x^2 + a_{n-1}x$ равны. То есть $a_1a_2a_3 = a_{n-3}a_{n-2}a_{n-1}$? Пусть начало первой группы будет F_1, а начало второй группы - F_2 и их длина len.

Как это проверить? Я буду рассматривать ту самую точку x_0, какое-то случайное значение. Я насчитаю 2 массива:
h(0): a_0
h(1): $a_0x_0 + a_1$
h(2): $a_0x_0^2 + a_1x_0 + a_2$
h(3): $a_0x_0^3 + a_1x_0^2 + a_2x_0 + a_3$
...
h(n-1): $a_0x_0^{n-1} + a_1x_0^{n-2} + a_2x_0^{n-3} + ... + a_{n-2}x_0 + a_{n-1}$
h(n): $a_0x_0^n + a_1x_0^{n-1} + ... + a_{n-1}x_0 + a_n$

Данную операцию мы выполняем за O(N). Как мы используем это для решения задачи? Я хочу вычислить значение для полинома $a_1x^{n-1} + a_2x^{n-2} + a_3x^{n-3}$. Вот у меня есть такая штука: h(3): $a_0x_0^3 + a_1x_0^2 + a_2x_0 + a_3$. Кусочек $a_1x_0^2 + a_2x_0 + a_3$ мне интересен. Поэтому я h(0) умножаю на x_0^3 и вычитаю получившееся выражение из h(3). Теперь я сделаю те же дей-

ствия для последних чисел.

h(n-1): $a_0x_0^{n-1} + a_1x_0^{n-2} + a_2x_0^{n-3} + ... + a_{n-3}x_0^2 + a_{n-2}x_0 + a_{n-1}$. Мне нужно добыть $a_{n-3}x_0^2 + a_{n-2}x_0 + a_{n-1}$. Мне нужно h(4), то есть без последних трёх. Он равен $h(4) = a_0x_0^{n-4} + a_1x_0^{n-5} + a_2x_0^{n-6} + ...$ Ну а теперь нам нужно умножить это на x_0^3 и вычесть из h(n-1).

Теперь выведем общий случай: как же сравнивать два произвольных кусочка полинома? Что, если $a_1x_0^2 + a_2x_0 + a_3$ и $a_{n-3}x_0^2 + a_{n-2}x_0 + a_{n-1}$ дают одинаковое значение в точке x_0? Я могу уверено утверждать, что $a_1 = a_{n-3}$, $a_2 = a_{n-2}$ и $a_3 = a_{n-1}$.

Идея сравнения: я беру и считаю $h(F_1)$, где $F_1 = 1$ (по коэффициенту). Далее записываю два выражения: $h(F_1 + len - 1) - h(F_1 - 1) * x^{len}$ и $h(F_2 + len - 1) - h(F_2 - 1) * x^{len}$, и если они равны, то и коэффициенты совпадают, а это значит, и полиномы тоже равны. Но тут возникает сложность. Когда мы сталкиваемся с такими большими степенями, скорее всего числа будут очень большие, а также x_0 ни в коем случае нельзя брать вещественным или какое-то скучное значение, по типу единицы, потому что полиномы плохо себя ведут при таких значениях и, наконец, если мы берем какое-то большее значение, у нас функция начинает очень быстро расти, поэтому скорее всего считать мы ее будет по модулю, как это делали в хеш-таблицах, но у вас могут возникнуть всякие неприятные проблемы.

Зачем мы всё это делали? Это позволит нам сравнивать кусочки подстрок. Например, строка bacdabacd. Занумеруем их: 213412134. Степень полинома здесь 8.

$2x^8 + x^7 + 3x^6 + 4x^5 + x^4 + 2x^3 + x^2 + 3x^1 + 4x^0$ в точке $x_0 = 10$
Выпишу h(i) для этого полинома:
h(0): 2
h(1): 21
h(2): 213
h(3): 2134
h(4): 21341
h(5): 213412
h(6): 2134121
h(7): 21341213
h(8): 213412134

Предположим я хочу сравнить подстроки acd и acd. В данном случае $F_1 = 1$ при нумерации с нуля, $F_2 = 6$ и len $= 3$. Вернёмся к нашим формулам: $h(F_1+len-1)-h(F_1-1)*x^{len}$ и $h(F_2+len-1)-h(F_2-1)*x^{len} \iff$ $h(3)-h(0)*1000 = 134$ и $h(8)-h(5)*1000 = 134$. Следовательно, эти подстроки равны. Или если я захочу сравнить acd и bac, то значения хеша будут: $h(3) - h(0) * 1000 = 134$ и $F_2 = 5$, значит, $h(7) - h(4) * 1000 = 213$. Но $213 \neq 134$, а значит, что строки не совпадают. Теперь решим проблемы, которые возникли. Если вы запрограммируете прямо так, то у вас не будет проблем для не очень длинных длинных строк, потому что для очень больших строк операции начинают выполняться не за O(1), а за O(кол-во цифр в этом числе) что довольно медленно. Поэтому на маленьких данных этот алгоритм будет работать за O(N), а на больших - за $O(N^2)$. Поэтому числа нам надо брать по модулю какого-то волшебного числа P. Но возникнут проблемы с вычитанием, поэтому перепишем нашу формулу.

$$(h(F_1+len-1)+h(F_2-1)*x^{len})\%P == (h(F_2+len-1)+h(F_1-1)*x^{len})\%P$$

Но внимательные слушатели скажут, что такой алгоритм всё еще не работает за O(1), и будут правы. Поэтому, нам нужно перезаписать метод возведения в степень:

```
s = s[1]...s[n]
h[0] = 0
x[0] = 0
x_0 = 10
p = 10**18
for i in range(1, n + 1):
    h[i] = (h[i - 1] * x_0 + ord(s[i])) % P
    x[i] = (x[i - 1] * x_0) % P
```

10.2 Применение хешей

Задача 1. Пусть дана строка abcabcab. Нужно найти самый короткий префикс, повтор которого даст всю нашу строку. С помощью хешей мы можем поиск до O(1). Теперь немного поменяем задачу: нам нужно найти самый длинный префикс, совпадающий с суффиксом. Опишем это формально: Пусть первые k символов не входят в суффикс и последние k символов не входят в префикс.
$\forall i > k : s[i] = s[i - k]$, где i - это i, попадающее в префикс. Теперь мы можем сравнивать подстроки за O(1).

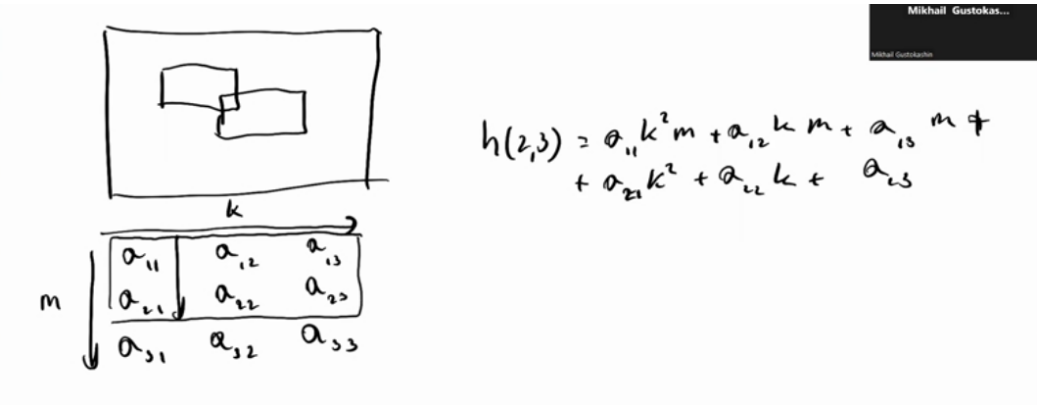
10.3 Z-функция

Пусть дана z-функция. Она определена для каждой позиции в строке и означает максимальную по длине подстроку, которая начинается с этой позиции и совпадает с префиксов.

Задача 2. Пример: abacaba. Давайте подсчитаем в этом примере z-функцию. То есть значения такой строки будут иметь вид 0010301. Как ее считать? Воспользуемся бинпоиском. Если они совпадают, то меняем левую границу, если не совпадают, то меняем правую границу. Такой алгоритм будет работать за $\theta(NlogN)$. Где применяется такая функция? Например, она может применяться для поиска всех вхождений подстроки в строку.

10.4 Двумерные хеши

Мы научимся сравнивать два подпрямоугольника в табличке. Пусть у нас есть коэффициенты в табличке. Строки обозначим за m, столбцы за k. Тогда хеш прямоугольничка будет обозначаться как h(m,k), где m - количество входящих строк и k - кол-во входящих столбцов. Первую строку мы можем рассматривать как одномерный хеш. Как на картинке:



Теперь я хочу выразить хеш прямоугольничка по аналогии с одномерными хешами.
$h(i,j) = h(i-1,j) \times m + h(i,j-1) \times k + a_{ij} - h(i-1,j-1) \times km$

Теперь научимся сравнивать два произвольных подпрямоугольника в нашей таблице. Так же, как и со строками, нам нужно научится получать в чистом видео наш прямоугольничек. Нам нужно знать минимальную и

максимальную координату нашего прямоугольника. В чём заключается идея? Мы знаешь хеш уголка (i_{min}, j_{min}). Для того, чтобы убрать лишние степени, которые там торчат, мы берем хеш от уголка (i_{min}, j_{max}) и умножаем его на $m^{i_{max}-i_{min}}$, чтобы привести степени в соответствие. Затем мы вычитаем из нашего хеша вычитаем хеш уголка с координатами (i_{max}, j_{min}), домноженный на $k^{j_{max}-j_{min}}$. Но получилось, что мы вычли уголок (i_{min}, j_{min}) два раза, поэтому мы его снова добавляем и умножаем на $k^{j_{max}-j_{min}}m^{i_{max}-i_{min}}$.

Спасибо, что дочитали до конца!