Жадные алгоритмы

Многие современные задачи связаны с нахождением оптимального значения какого-нибудь параметра. Например:

- найти такой маршрут из Москвы, чтобы путешественник посетил Владивосток, Якутск и Иркутск в любом порядке и при этом потратил на перелёты минимальную сумму денег;
- выбрать блогеров для размещения рекламы с ограниченным бюджетом так, чтобы её увидело максимальное число людей;
- составить план расширения сотовой сети таким образом, чтобы покрыть связью все необходимые области, используя при этом минимальное количество вышек.

Задачи нахождения оптимального значения называются «экстремальными», а процесс их решения — «оптимизацией».

Для оптимизации часто бывает необходимо считать интегралы и решать дифференциальные уравнения. Но в некоторых задачах можно найти экстремум, не обращаясь к серьёзной математике — с помощью жадных алгоритмов и динамического программирования.

О том, как это делается, мы расскажем в этом спринте. И начнём с жадных алгоритмов.

Что такое жадность в алгоритме

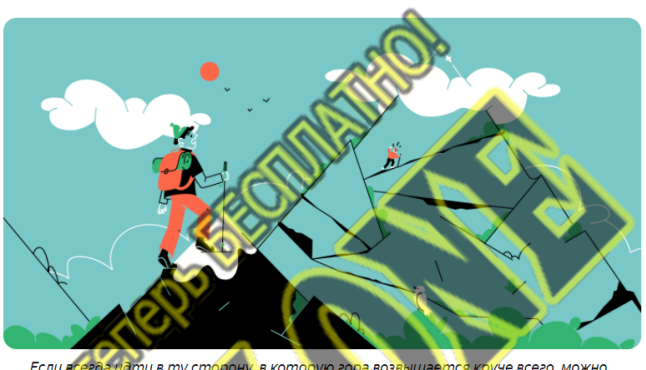
Алгоритмы, о которых мы будем сейчас говорить, похожи на жадного до игрушек малыша: получив от родителей деньги на карманные расходы, он сразу же тратит их на какую-нибудь новую игрушку. Раз за разом он выбирает недорогую безделушку, которую может купить прямо сейчас и которая надоест ему уже завтра. Хотя он мог бы накопить побольше денег и потратить их на игру, которая увлекала бы его в течение долгого времени.

Жадные алгоритмы работают по такому же принципу: на каждом шаге они выбирают решение, оптимальное по определённому критерию (в случае с малышом — ценовая доступность игрушки прямо сейчас), не заглядывая в будущее.

В контексте данного урока мы будем называть **тактикой** действия, которые являются оптимальными в настоящий момент, безотносительно к перспективе; а **стратегией** — действия, которые могут не влиять на ситуацию сейчас или даже ухудшать её, но в долгосрочной перспективе приводят к наилучшему результату.

Бывают ситуации, когда оптимальное тактическое решение (например, всегда покупать максимально свежие овощи), совпадает с оптимальным стратегическим решением (сделать на ужин максимально свежий салат). В таких случаях применение жадных алгоритмов будет очень эффективным: они быстро работают, потому что не тратят времени на оценку последствий своего выбора.

Однако при подъёме на гору лучшее, на первый взгляд, тактическое решение — идти в сторону самого крутого подъёма — не поможет в достижении стратегической цели — оказаться на вершине самой высокой горы. Ведь иногда, чтобы забраться на самую высокую вершину, надо пройти по равнине — мимо нескольких менее высоких гор.



Если всегда идти в ту сторону, в которую гора возвышается круче всего, можно найти самую высокую точку этой горы (то есть локальный максимум). Но для того, чтобы найти самую высокую гору (то есть глобальный максимум), такая стратегия не подойдёт

Эта часть спринта может показаться слишком простой. Но не торопитесь с выводами! При кажущейся простоте жадные алгоритмы затрагивают сложные и важные темы.

Гоша и сувениры

В последний день отпуска на Алгосах Гоша пришёл в сувенирный магазин, чтобы выбрать для своих друзей маленькие подарки из поездки — магниты для холодильника. Все они выглядят очень привлекательно, но цена у них разная. У Гоши с собой есть только X алгосских франков, и он хочет купить максимально возможное количество магнитов, чтобы порадовать как можно больше друзей. Какую тактику в этой ситуации следует выбрать Гоше?

Все магниты кажутся Гоше красивыми, поэтому единственный критерий, по которому он может их выбирать, это цена. Так как задача Гоши — купить как можно больше магнитов, то в приоритете будут самые дешёвые из них. Ведь чем меньше стоит каждый купленный магнит, тем больше денег остаётся на покупку оставшихся.

Алгоритм будет выглядеть так:

1. Разложить магниты в порядке увеличения стоимости.
2. До тех пор, пока хватает денег и не кончились магниты, покупать самый дешёвый магнит.

Скорее всего, под конец у Гоши останется какое-то количество алгосских франков, но если он при этом не сможет купить даже самый дешёвый магнит из оставшихся, то и на более дорогой ему денег не хватит.

Алла
Кажется, мы все иногда интуитивно ведём себя, как жадные алгоритмы.

Тимофей
Верно. Кстати, вы уже решали похожие задачи.

Рита
А я вспомнила про сортировку выбором, о которой читала после спринта с сортировками. Чтобы упорядочить массив, надо каждый раз просматривать ещё не отсортированную часть и выбирать наибольший или наименьший элемент — в зависимости от желаемого порядка сортировки.

Этот алгоритм перемещает найденный элемент в конец уже отсортированной части массива и не просчитывает, получится ли отсортировать весь массив или нет. Благодаря такому подходу после i-ого шага в отсортированной части массива будет i самых больших или самых маленьких значений. Любой элемент, расположенный правее этих i значений, будет или не больше, или не меньше любого из них.

В прошлом спринте мы разобрали несколько полезных алгоритмов. Как вы думаете, какие из них работают по жадному принципу?

- [ ] Алгоритм Дейкстры — для построения дерева оптимальных маршрутов.

На каждом шаге мы выбирали ещё не посещённую вершину, которая удалена от стартовой не больше, чем любая другая не посещённая вершина.

- [x] Алгоритм Прима — для построения минимального остовного дерева.
- [ ] Поиск в глубину DFS.
- [ ] Поиск в ширину BFS.

Решите задачу «Биржа».

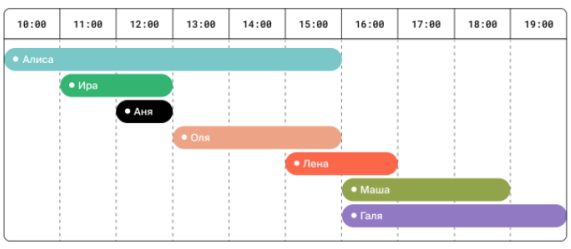
Вперёд

Задача о расписании

Давайте рассмотрим ещё одну задачу, которая решается жадностью.

Алла приехала в город своего детства и хочет встретиться с подругами. Времени у неё мало — всего один день, а подруг много — и каждая зовёт Аллу на какое-нибудь мероприятие с точным временем начала и конца. Чтобы ничего не забыть, Алла внесла в календарь все события, на которые её пригласили подруги.

Календарь встреч



Получилось целых семь встреч за один день! Увы, ни приходить позже, ни уходить раньше Алла на эти встречи не сможет, так как она очень ответственная и пунктуальная девушка. Как же Алле выбрать, на какие мероприятия пойти, а какие пропустить — чтобы увидеть при этом как можно больше своих подруг?

Варианты решений

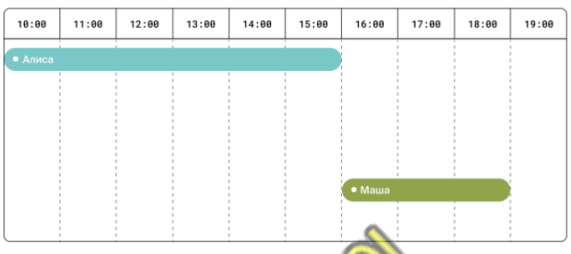
Если Алла не хочет терять ни минуты , она пойдёт на то мероприятие, которое начинается раньше других.

Для этого ей нужно отсортировать все встречи по времени начала. При такой сортировке Алла:

- пойдёт на первое мероприятие из списка,
- пропустит все мероприятия, которые начнутся раньше, чем закончится то, на котором она находится сейчас.

После окончания встречи Алла будет выбирать новую по тому же алгоритму.

Сортировка по времени начала встречи



Алла

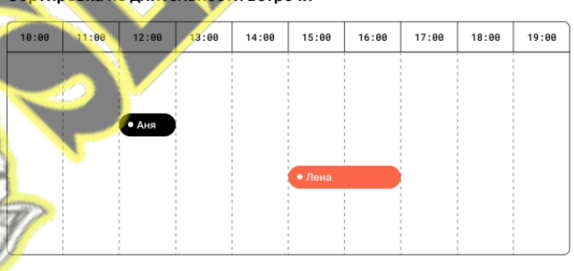
При такой стратегии у меня будет целых 9 часов встреч, а увижусь я только с Алисой и Машей. Мне кажется, это не самое лучшее решение. Попробую составить своё расписание иначе.

А что если я буду выбирать сначала самые короткие встречи, а потом те, что длиннее?

Рита

Гоша по этому же принципу смог купить максимально возможное количество сувениров. Вдруг и у тебя так получится встретиться с максимально возможным числом людей?

Сортировка по длительности встречи



Опять плохо! Гоше было проще: ведь покупка одного магнита не мешала ему покупать другие. С мероприятиями так не получается.

Вот три самых коротких встречи: Аня (1 час), Ира (2 часа), Лена (2 часа). Но встречи с Ирой и с Аней пересекаются, поэтому Алла пойдёт только на одну из них.

Итого в расписании Аллы останется только 2 коротких встречи, и между ними не получится вставить ничего другого. Но если Алла отменит мероприятие с Леной, она сможет встретиться и с Олей, и с Машей. А это уже целых три встречи вместо двух!

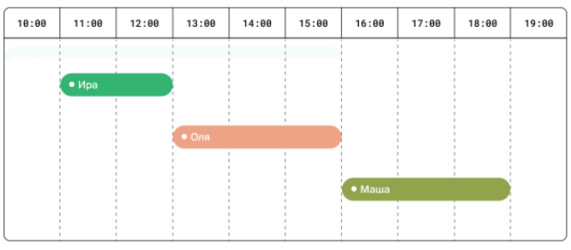
Рита

У меня есть идея! Попробуй выбирать мероприятие, которое ещё не началось, но закончится раньше остальных.

Алла

Хм... Это какая-то совсем непонятная жадность, но результат верный. Так я действительно смогу посетить целых три встречи.

Сортировка по правой границе



Рита

Объяснить готовое решение можно так: чем раньше закончится мероприятие, тем больше времени для других встреч у тебя останется!

Алла

Действительно! Спасибо за помощь!

Решите задачу «Расписание».

Вперёд

Задача о рюкзаке

Тимофей

Ну раз все разъехались по отпускам, то и я отправлюсь отдохнуть. Как раз давно хотел пойти в горы с палаткой. А пока буду собирать рюкзак, расскажу вам ещё об одной задаче.

Перед каждым туристом рано или поздно встаёт вопрос: какую еду брать с собой в поход. С одной стороны, продуктов должно хватить на все дни путешествия и желательно, чтобы они были максимально сытные. С другой — еда не должна быть слишком тяжёлой и занимать много места, иначе рюкзак окажется неподъёмным.

Тимофей уже положил в свой рюкзак обязательное снаряжение и необходимый минимум тушёнки. Ведь без неё и поход не поход! Осталось выбрать крупу. Её Тимофей решил взять не более пяти килограммов — чтобы рюкзак не получился слишком тяжёлым.

Жадное решение

В кладовке у Тимофея лежит несколько килограммов разной крупы: гречневой, рисовой, пшённой и манной. У каждой из них есть энергетическая ценность. В походе очень важно, чтобы еда давала много энергии: чем сытнее поешь, тем больше у тебя будет сил и активнее пройдёт день.

Тимофей знает, как собрать наиболее сытный набор крупы, но хочет, чтобы ребята сами нашли алгоритм решения этой задачи. Важное уточнение: пачки с крупой можно вскрывать и пересыпать необходимое количество каши в контейнеры.

Рита

Так ведь эта задача почти не отличается от ситуации, когда Гоша выбирал магниты! Он хотел купить максимум сувениров за минимум денег, а тебе нужно положить в рюкзак максимум калорий при минимуме грузоподъёмности.

Тимофей

Да, задачи действительно очень похожи. Поделишься своим решением?

КРУПА (вид)	КАЛОРИЙНОСТЬ (ккал/100 г)	ЗАПАС (кг)
Гречневая	330	4
Рисовая	290	2
Пшённая	305	3
Манная	320	2

Калорийность крупы рассчитывается в сухом виде. После того как крупу сварят, её вес увеличится, и в 100 г каши будет существенно меньше калорий, чем в 100 г сухого продукта

Решение такое: отсортируем крупы по уменьшению энергетической ценности и будем брать максимально возможное количество самой сытной крупы до тех пор, пока либо не исчерпается лимит в пять килограмм, либо не кончится крупа. Во втором случае возьмём следующую по калорийности крупу и повторим те же действия.

Итого мы возьмём четырёхкилограммовую пачку гречки и отсыпем в контейнер 1 кг манки. Получим такую энергетическую ценность: $3300 \cdot 4 + 3200 \cdot 1 = 16400$ ккал.

Решите задачу «Золотая лихорадка».

Тимофей

А теперь представьте, что пересыпать крупы в контейнеры нельзя. Вы можете либо взять пачку целиком, либо не брать её вовсе. Как тогда изменится алгоритм?

Рита

Зачем ему меняться? Я буду жадно набирать пачки с самой калорийной крупой, пока это будет возможно.

Тимофей

Увы, твой алгоритм даст некорректный ответ.

Жадный алгоритм возьмёт в итоге только четырёхкилограммовую пачку гречки, калорийность которой будет 13 200 ккал. Но если вместо этого взять 2 кг манки и 3 кг пшёнки, то их суммарная энергетическая ценность окажется больше: $3200 \cdot 2 + 3050 \cdot 3 = 15550$ ккал.

Как видите, жадный алгоритм не может посмотреть в будущее и отказаться от высококалорийной гречки — ради потенциально более выгодного дуэта манки и пшёнки.

Как доказать, что жадного решения нет

Гоша

А может, какое-то жадное решение всё-таки можно найти?

Тимофей

К сожалению, нет. В такой формулировке задача о рюкзаке жадным алгоритмом не решается.

Про каждое отдельное жадное решение можно сказать, является ли оно корректным или нет. Но не существует такого множества решений, перебрав которое, можно было бы сказать: «Так как все решения оказались некорректными, то жадного решения для этой задачи нет». Поэтому, строго говоря, доказать отсутствие жадного решения невозможно.

Алла

Получается очень сложно. Когда же надо бросить поиски жадного решения и переключиться на поиск решения другого типа?

Тимофей

Всегда надо проверять два-три решения, которые первыми приходят на ум. Обычно часть из них будет жадными. Если ни одно из этих решений не даёт правильный ответ всегда (я потом расскажу, как это проверить), стоит поискать другой класс алгоритмов.

Алла

А может, есть какой-то список задач, которые совершенно точно не решаются жадными алгоритмами? Хорошо бы с ним ознакомиться, чтобы не терять потом времени на поиск несуществующего решения.

Тимофей

Да, такой список есть. В нём указаны классические задачи, которые жадностью не решить: задача о рюкзаке, задача коммивояжёра и даже алгоритм оптимальной последовательности шагов для «пятнашек».

В таких задачах важны все части формулировки. Например, если мы сможем брать крупы частями, а не целиком, то задача о рюкзаке прекрасно решится с помощью жадной стратегии.

Точное решение задачи о рюкзаке

Итак, нам нужно положить в рюкзак не более пяти килограммов крупы с максимальной общей энергетической ценностью. Есть некоторое количество возможных комбинаций круп из кладовки Тимофея — и это множество не бесконечно. А значит, мы можем перебрать все варианты, посчитать для каждого энергетическую ценность и решить, какой из них подходит нам больше всего.

Нам надо перебрать все наборы круп. В каждом наборе мы решаем: берём мы данную упаковку или нет. Если берём — в соответствующую ячейку ставим 1, если нет –– 0. Таким образом нам надо перебрать все последовательности из 0 и 1 длины N, где N — количество разных круп. Таких последовательностей 2^N.

0 – не берём 1 – берём				ВЕС	КАЛОРИЙНОСТЬ	КОММЕНТАРИЙ
Гречневая	Рисовая	Пшённая	Манная			
0	0	0	0	0	0	Допустимо, но не оптимально
0	0	0	1	2	6.400	Допустимо, но не оптимально
0	0	1	0	3	9.150	Допустимо, но не оптимально
0	0	1	1	5	15.550	Лучшее решение
0	1	0	0	2	5.800	Допустимо, но не оптимально
0	1	0	1	4	12.200	Допустимо, но не оптимально
0	1	1	0	5	14.950	Допустимо, но не оптимально
0	1	1	1	7	21.350	Перевес
1	0	0	0	4	13.200	Допустимо, но не оптимально
1	0	0	1	6	19.600	Перевес
1	0	1	0	7	22.350	Перевес
1	0	1	1	9	28.750	Перевес
1	1	0	0	6	19.000	Перевес
1	1	0	1	8	25.400	Перевес
1	1	1	0	9	28.150	Перевес
1	1	1	1	11	34.550	Перевес

В половине случаев получившийся набор круп не влезает в рюкзак. Из оставшихся восьми наборов мы выбираем тот, в котором калорийность наибольшая

Приближённое решение

Программисты часто применяют жадные алгоритмы для быстрого получения достаточно хорошего приближённого решения — вместо того, чтобы тратить время на вычисление точного ответа.

Однако отклонение жадного решения от точного может получиться сколь угодно большим. В примере с рюкзаком Тимофея предложенный жадный алгоритм собрал набор из разных круп с калорийностью 13 200 ккал, а точное решение получило 15 550 ккал. Разница между ними — 2 350 ккал, то есть жадное решение сработало почти на 15% хуже, чем точное.

Если мы будем сравнивать стоимость обедов в соседних кафе, то разница в 15% может показаться некритичной. Но при покупке квартиры колебание цены на 15% будет весьма ощутимым.

Важно понимать, что 15% — не верхняя граница различия. Точное и жадное решения могут различаться и в 10 раз, и даже больше. Заранее предугадать погрешность вычислений не получится — она всегда зависит от входных данных.

Решая задачи жадным методом, надо различать ситуации:

- есть доказательство, что жадный алгоритм даст точное решение;
- такого доказательства нет — жадный алгоритм экономит наше время и даёт приближённое решение.

Во втором случае мы рекомендуем для решения задачи применять несколько жадных алгоритмов (например, в ситуации с расписанием Аллы мы попробовали три разных стратегии) и для каждого набора входных данных брать ответ того алгоритма, который получил более выгодное решение (наибольшее количество встреч — для Аллы, максимально калорийный набор круп — для Тимофея).

Рита

А как понять, в какой я ситуации нахожусь? Как доказать, что выбранная стратегия даст точное решение, а не приближённое?

Тимофей

В этом нам поможет метод математической индукции. О нём я расскажу в другой раз.

Вперёд

Динамическое программирование. Методы формальных доказательств

На этом курсе мы уже рассмотрели много алгоритмов и на простых примерах объяснили, почему тот или иной алгоритм работает или нет. Однако бывает, что простое объяснение красиво выглядит и приятно читается, но в него закралась ошибка. Например, так было у Аллы, которая думала, что если она будет выбирать встречи по увеличению их продолжительности, то сможет посетить больше встреч.

Для того чтобы не обмануться, разработчик должен уметь проверять своё решение тестами и доказывать его корректность, используя методы формальных (или математических) доказательств.

Таких методов довольно много. Мы рассмотрим только некоторые из них. Но прежде, чем доказывать утверждения, поговорим о том, как их опровергать.

Опровержение утверждений

Математические утверждения очень похожи на задачи в системе Яндекс.Контест. Если ваше решение не проходит хотя бы один тест, оно считается неверным. Даже если на других 100 тестах ваше решение выдаёт правильный ответ. Так же и с математическими теоремами: доказать утверждение сложно, а опровергнуть легко — достаточно привести один контрпример.

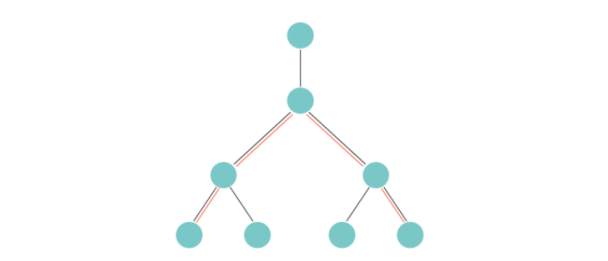
Рассмотрим следующее утверждение:

> Если в дереве каждый лист находится на расстоянии n от корня, то максимальная длина пути между двумя листьями равна $2 \cdot n$.

На первый взгляд, утверждение кажется верным, и мы можем привести много примеров деревьев, где это верно:

Все листья в приведённых деревьях расположены на удалении 3 от корня. И действительно, самый длинный путь между листьями в каждом из деревьев имеет длину 6. Красным цветом помечен один из вариантов пройти от одного листа до другого самым длинным путём

Однако нам следует искать не подтверждения этой гипотезы, а опровержения. И мы такое опровержение найдём:



Хотя у этого дерева все листья находятся на удалении 3 от корня, самый длинный путь между листьями имеет длину 4, а не 6, как мы предполагали. Этот контрпример опровергает гипотезу

Мы привели контрпример и таким образом показали, что утверждение «Если в дереве каждый лист находится на расстоянии n от корня, то максимальная длина пути между двумя листьями равна $2 \cdot n$» — неверно.

Теперь вернёмся к методам доказательств утверждений.

Метод полного перебора

Если множество значений, которые могут быть подставлены в утверждение, `конечно`, то можно проверить утверждение для каждого значения. Если для всех возможных значений утверждение корректно, то это корректное утверждение.

Таким способом можно доказать, что у всех учеников 10 «А» класса Алгосского лицея есть аттестат за 9 класс:

1. Опрашиваем всех учеников 10 «А» класса.
2. Если у всех опрошенных есть аттестат за 9 класс, значит, не существует такого ученика 10 «А» класса данной школы, у которого нет аттестата за 9 класс.

Когда мы доказываем что-то методом полного перебора, важно работать только с конечными множествами и не пропустить при проверке ни одного значения.

Метод прямого доказательства

Прямое доказательство используется, если из изначального высказывания через цепочку неоспоримых следствий следует итоговое заключение.

Например, докажем, что если целое число X делится на 6, то оно делится и на 2, и на 3.

1. Изначальное высказывание: «Число X делится на 6». Считаем, что оно истинно.
2. Если X делится на 6, это значит, что существует целое число Y, такое что $X = Y \cdot 6$.
3. Но так как $6 = 3 \cdot 2$, то $X = Y \cdot 3 \cdot 2$.
4. Раз так, то $X = (Y \cdot 2) \cdot 3 = (Y \cdot 3) \cdot 2$. Следовательно, X делится и на 3, и на 2.

Именно этот метод вы чаще всего встречали в школе на уроках геометрии.

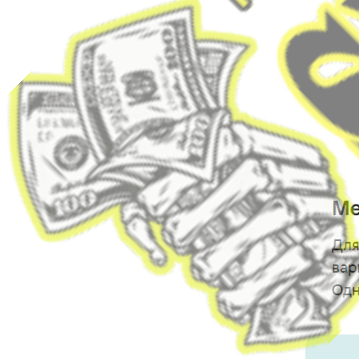
Метод доказательства от противного

Для того чтобы доказать утверждение от противного, необходимо предположить, что изначальное высказывание — ложно, а затем доказать каким-либо другим формальным методом, что при таком предположении становится неверно некоторое утверждение, которое заведомо верно. Таким образом, мы будем вынуждены признать, что изначальное высказывание истинно.

Пусть дан набор попарно различных чисел $\{a_i\}_{i=1\dots n}$ и нужно найти пару чисел a_j, a_k с наименьшим расстоянием между ними, то есть наименьшим модулем разности $|a_j - a_k|$. Рассмотрим следующее решение: отсортируем набор чисел по возрастанию. Переобозначим числа их позициями в возрастающем списке: $b_1 < b_2 < \dots < b_n$. Переберём только те пары чисел, которые в отсортированном списке идут подряд: b_i, b_{i+1}. Возьмём пару с наименьшим расстоянием среди таких пар, она будет иметь наименьшее расстояние и среди всех пар.

Докажем, что это даст нам оптимальное решение.

1. Предположим, что это не так, то есть существует пара чисел b_j, b_k такая, что расстояние между ними меньше найденного, хотя они в отсортированном списке расположены не подряд.
2. В паре порядок элементов не важен, поэтому договоримся, что $j < k$. Поскольку элементы идут не подряд, $j + 1 \neq k$. А значит существует целое число $j + 1$, расположенное между j и k. Поскольку ряд $\{b_i\}_{i=1\dots n}$ упорядочен, а все элементы в нём попарно различны, то $b_j < b_{j+1} < b_k$. Вычтем b_j из всех частей неравенства и немного преобразуем. Получим, что расстояние $b_{j+1} - b_j < b_k - b_j$. Это противоречит нашему предположению, что расстояние $b_k - b_j$ между двумя несмежными элементами может оказаться наименьшим. Раз предположение оказалось неверным, то пара элементов, расстояние между которыми наименьшее из возможных, в отсортированном массиве непременно будет располагаться в соседних ячейках.



Метод доказательства по индукции

Для того чтобы проверить утверждение методом математической индукции, все варианты входных значений должны быть последовательно пронумерованы. Однако сделать это возможно не всегда.

> В отличие от метода полного перебора количество входных значений для метода математической индукции может быть бесконечным. К примеру, всё множество натуральных чисел: мы всегда можем назвать число больше предыдущего, но это не мешает нам пронумеровать по порядку все числа.

Метод доказательства по индукции подразумевает, что сначала проверяется истинность утверждения с номером $n = 1$. Это утверждение называют «базой индукции». Обычно в качестве базы берут легко доказуемое утверждение (либо несколько утверждений).

Далее рассматривается «шаг индукции». Для этого делают предположение, что доказано утверждение под номером n. Исходя из этого предположения доказывают, что будет верно и утверждение под номером $n + 1$.

Имея эти две компоненты, можно доказать утверждение для любого натурального n:

- Сначала мы воспользуемся базой индукции, которая говорит, что утверждение верно при $n = 1$.
- Затем при помощи шага индукции доказываем, что раз утверждение верно для $n = 1$, то оно верно и для $n = 2$.
- После этого шаг индукции приведёт нас к верности утверждения при $n = 3, 4, \dots$ и так далее.

Если i-ое утверждение обозначить за $H(i)$, то можно компактно написать принцип математической индукции так:

$$\left.\begin{array}{r} H(1) \text{ верно} \\ \text{если } H(i) \text{ верно для } i \leqslant n, \text{ то } H(n+1) \text{ верно} \end{array}\right\} \Rightarrow$$
$$H(i) \text{ верно для любых } i \in \mathbb{N}$$

Рассмотрим небольшой пример. При оценке сложности алгоритмов нам часто приходилось суммировать числа $1 + 2 + 3 + \dots + n$. Докажем при помощи метода математической индукции, что эта сумма равна $\frac{n \cdot (n+1)}{2}$.

Формула, которую мы пытаемся доказать, должна быть верна для всех натуральных чисел. Но мы её разобьём на ряд утверждений $H(i)$.

Утверждение $H(i)$ гласит: $1 + 2 + 3 + \dots + i = \frac{i \cdot (i+1)}{2}$.

В данном случае i выступает и членом формулы, и номером утверждения. Это типичная ситуация.

1. Сначала мы докажем, что формула верна для $i = 1$, то есть что верно утверждение $H(1)$, база индукции. Для этого достаточно проверить, что $1 = \frac{1 \cdot (1+1)}{2}$.
2. Шаг индукции: пусть $H(i)$ верно для всех i от 1 до n. Зная это, нам нужно доказать, что верно утверждение $H(n+1)$. Распишем сначала уже доказанные утверждения:

$$\begin{cases} H(1): \ 1 = \frac{1 \cdot (1+1)}{2} \\ H(2): \ 1 + 2 = \frac{2 \cdot (2+1)}{2} \\ H(3): \ 1 + 2 + 3 = \frac{3 \cdot (3+1)}{2} \\ \dots \\ H(n): \ 1 + 2 + 3 + \dots + n = \frac{n \cdot (n+1)}{2} \end{cases}$$

Теперь распишем утверждение, которое мы хотим доказать: $H(n+1): \ 1 + 2 + 3 + \dots + n + (n+1) = \frac{(n+1) \cdot (n+2)}{2}$

В данном случае для доказательства нам будет достаточно воспользоваться только утверждением $H(n)$, чтобы упростить часть суммы:

$$\underbrace{1 + 2 + 3 + \dots + n}_{\text{упростим при помощи } H(n)} + (n+1) = \frac{n \cdot (n+1)}{2} + (n+1) = \frac{(n+1) \cdot (n+2)}{2}$$

Видим, что вычисления привели нас к желаемому результату. Таким образом мы доказали, что $H(n+1)$ верно.

3. Раз база индукции верна и шаг индукции доказан, то утверждение индукции верно для любых i. Можно представить доказательство как цепочку логических выводов:

$$\begin{cases} H(1) \text{ верно (база индукции)} \\ H(1) \text{ верно} \Rightarrow H(2) \text{ верно} \\ H(1), H(2) \text{ верны} \Rightarrow H(3) \text{ верно} \\ H(1), H(2), H(3) \text{ верны} \Rightarrow H(4) \text{ верно} \\ \dots \\ H(1), H(2), H(3), \dots, H(n) \text{ верны} \Rightarrow H(n+1) \text{ верно} \\ \dots \end{cases}$$

Резюме

Мы рассмотрели 4 популярных способа, которыми можно доказывать теоремы математики и Computer Science. Чтобы убедиться в корректности алгоритма, вы должны сформулировать его в виде чёткого математического утверждения, а затем доказать его.

Для примера сформулируем формально задачу о составлении наилучшего расписания. Пусть есть множество интервалов $I = \{(s_i, f_i) | i \in 1 \dots n\}$. Обозначим за m максимальный размер такого подмножества I, что никакие два интервала этого подмножества не пересекаются.

Утверждается, что алгоритм, итеративно выбирающий из I по одному такие интервалы (s_j, f_j), что они имеют наименьшую правую границу f_i, и при этом не пересекают ранее взятые интервалы, сможет подобрать m интервалов. Тем самым мы получим одно из возможных «максимальных» расписаний. Чтобы убедиться в корректности алгоритма, следует доказать это утверждение.

Для этого можно воспользоваться методом математической индукции. Индукция будет идти по размеру множества интервалов I, то есть на каждом шаге индукции размер множества интервалов будет увеличиваться на 1. Для пустых множеств и множеств, состоящих из одного единственного интервала, утверждение тривиально: алгоритм просто выберет единственный интервал — и это и будет максимальное расписание. Таким образом, база индукции доказана.

Шаг индукции предполагает, что алгоритм работает корректно для любого множества интервалов, имеющего размер не больше n. На базе этого предположения следует доказать, что из $n + 1$ интервала алгоритм тоже составит расписание максимальной длины. Это доказательство оставим вам в качестве упражнения.

В задачах этого спринта вам придётся проверять корректность ваших алгоритмов. В разных задачах работают разные подходы. Вы можете попробовать проводить доказательство любым из описанных методов. Если не получается доказать утверждение о корректности алгоритма, попробуйте применить другой способ.

Иногда ваши алгоритмы будут содержать ошибку, и вы будете пытаться доказать некорректное утверждение. Если действовать по правилам, доказать некорректное утверждение невозможно, так что доказательство у вас придумать не получится. Но сама попытка доказать утверждение может навести вас на проблемные места и позволит лучше понять, в каких случаях ваш алгоритм работает некорректно.

Динамическое программирование. Алгоритм решения

Динамическое программирование — метод решения задачи путём разбиения её на задачи меньшего размера. Решение более сложной задачи складывается из решений более простых подзадач.

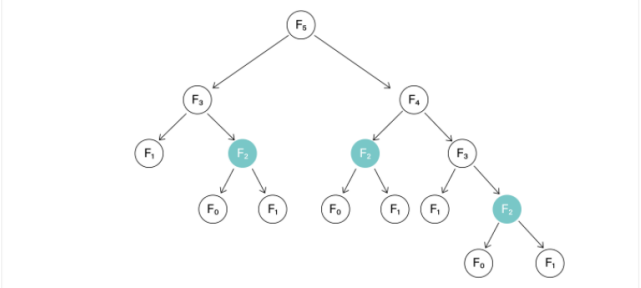
Алла

Но ведь ровно так же работают рекурсивные алгоритмы! В чём разница между рекурсией и динамическим программированием?

Тимофей

Всё верно, динамическое программирование — это рекурсия, вывернутая наизнанку. Сейчас объясню, что я имею ввиду.

В одном из прошлых уроков вы встречали задачу про числа Фибоначчи. Давайте вспомним, как вычислить i-ое число Фибоначчи: $F_i = F_{i-1} + F_{i-2}$ для всех $i \geq 2$, а $F_0 = F_1 = 1$.



Если мы хотим вычислить пятое число Фибоначчи, то в процессе нам придётся трижды вычислять второе число Фибоначчи. А для вычисления F_6 пришлось бы вычислить F_2 уже пять раз. Для вычисления F_{40} повторить вычисление F_2 пришлось бы 63245986 раз

Используя рекурсию для решения этой задачи, мы вынуждены несколько раз вычислять одни и те же значения. Но можно поступить иначе:

- создадим массив для хранения уже посчитанных чисел Фибоначчи;
- и прежде, чем вычислять очередное F_k, будем проверять, есть ли оно в нашем массиве или ещё нет.

Так мы сэкономим время, ведь каждое число Фибоначчи будет посчитано ровно один раз и затем окажется в хранилище, которое принято называть кэшем. Этот вид оптимизации называется «ленивая динамика».

Алла

Ух ты! Это поможет нам не переполнить стек вызовов и посчитать даже очень большие числа Фибоначчи! А почему динамика ленивая? И чем она отличается от нелениво?

Тимофей

Ленивая динамика будет что-то считать, только если пользователь запросил результаты вычислений. Кроме того, она никогда не будет делать одну и ту же работу дважды.

А нелениваяя, как ты её назвала, динамика не ждёт, когда у неё запросят значение функции. Она из условия задачи знает самое большое число Фибоначчи, которое у неё могут спросить, и последовательно вычисляет все остальные: сначала F_2, потом F_3 и так далее — до тех пор, пока не дойдёт до максимального.

Алла

Как эффективно решить конкретную задачу с помощью динамики, я поняла. А как в общем случае решать задачи динамического программирования?

Алгоритм решения

Для решения любой задачи методом динамического программирования надо ответить на пять вопросов.

1. **Что будет храниться в массиве `dp`?**

Название dp — сокращение от английского Dynamic Programming. В массив `dp` сохраняют ответы на подзадачи, сопоставляя индекс в массиве с условным номером подзадачи.

В примере с числами Фибоначчи массив заполнен таким образом, что $dp[i] = F_i$.

2. **Каким будет базовый случай для задачи?**

Базовый случай — это те значения, которые необходимо и достаточно вычислить вручную, чтобы программа, отталкиваясь от них, смогла вычислить всё остальное самостоятельно.

В задаче с числами Фибоначчи есть два базовых случая. Мы заранее знаем, что $F_0 = F_1 = 1$. То есть заполняем: $dp[0] = 1, dp[1] = 1$.

Базовый случай динамики всегда совпадает с базовым случаем рекурсивного решения этой же задачи.

3. **Каким будет переход динамики?**

Необходимо понять, как получить ответ для исходной задачи — зная ответы на подзадачи. В простых случаях для этого достаточно решить всего одну подзадачу. Допустим, программе нужно уметь вычислять степень числа 2^i. В этом случае формула перехода выглядела бы так:

$$dp[0] = 1;\ dp[i] = 2 \cdot dp[i-1]$$

Формулу перехода часто называют `рекуррентной` формулой.

Для чисел Фибоначчи на каждом шаге комбинируются две подзадачи. Но бывает и так, что комбинировать приходится много подзадач. Например, подзадачи всех возможных размеров.

4. **Каким будет порядок вычисления данных в массиве `dp`?**

На этом шаге надо решить, в каком порядке будет заполняться массив `dp`. По возрастанию индекса, по убыванию или более сложным образом.

Помимо порядка заполнения массива можно менять способ вычисления значения в массиве `dp`. Есть два похода к вычислению: динамическое программирование назад и динамическое программирование вперёд.

- Динамическое программирование назад подразумевает, что при вычислении очередного значения мы используем уже вычисленные. То есть мы как будто оглядываемся назад и смотрим на уже проделанную работу.
  $F[i] = F[i-1] + F[i-2]$
- Динамическое программирование вперёд работает иначе. Алгоритм как бы забегает вперёд и оставляет заметки в разных ячейках. А потом, когда до них доходит очередь, обрабатывает эти ячейки, используя записи, сделанные ранее. Из-за таких заметок в какие-то моменты в некоторых ячейках массива может находиться не полный ответ на подзадачу, а только его часть.
  $F[i+1] += F[i]$
  $F[i+2] += F[i]$

5. **Где будет располагаться ответ на исходный вопрос?**

В какие ячейки надо посмотреть, чтобы получить ответ. В нашем примере в ячейках записаны сами числа Фибоначчи, значит, искомое значение F_n будет записано в ячейке $dp[n]$.

Но не всегда ответ на исходную задачу хранится прямо в ячейке массива `dp`. В некоторых случаях ответом может быть, например, минимальное значение в массиве `dp` или сумма всех значений в `dp`.

Разбор задачи

Алла стоит перед лестницей. На каждом шаге она может либо переходить на следующую ступеньку, либо прыгать через одну. Сколькими способами Алла может добраться до n-ой ступеньки?

Поочерёдно ответим на пять вопросов:

1. Что будет храниться в массиве `dp`? `dp[i]` — количество способов добраться до ступеньки с номером i.
2. Каким будет базовый случай для задачи? $dp[1] = 1$, ведь до 1 ступеньки можно добраться всего одним способом. До 2 ступеньки — двумя способами: либо прыгнув сразу на вторую ступеньку, либо перейдя с первой ступеньки. $dp[2] = 2$.
3. Каким будет переход динамики? Есть 2 способа попасть на какую-то ступеньку. Либо сделать один шаг с предыдущей, либо прыгнуть сразу на 2 ступеньки — с предпредыдущей: $dp[i] = dp[i-1] + dp[i-2]$.
4. Каким будет порядок вычисления данных в массиве `dp`? Вычисляем очередное значение опираясь на посчитанные ранее значения. Для этого обновляем массив в порядке возрастания i. В нашем примере для того чтоб посчитать количество способов попасть на третью ступеньку суммируем количество способов добраться на вторую ступеньку и число способов попасть на 1 ступеньку;
5. Где будет располагаться ответ на исходный вопрос? Так как мы считаем количество способов попасть на ступеньку под номером n, ответ лежит в ячейке $dp[n]$.

Гоша

А я только сейчас заметил, что рекуррентная формула перехода один в один как в числах Фибоначчи!

Тимофей

Всё верно, они ловко замаскировались в этой задаче.

Решите задачу «Прыжки по лестнице».

Вперёд

Двумерная динамика

Тимофей

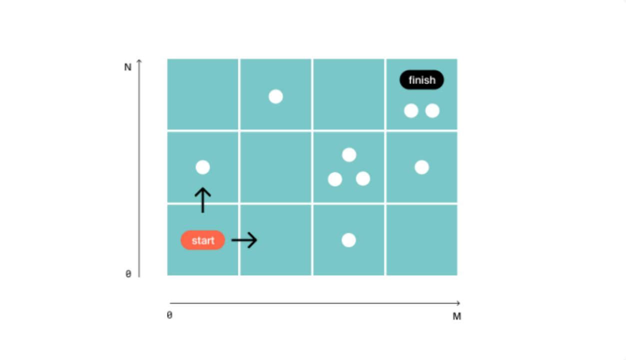
Давайте поиграем в игру! Допустим, у нас есть прямоугольное поле N на M. В каждой ячейке может быть какое-то количество фишек, а может и не быть ни одной. В начале игры мы находимся в левой нижней ячейке, и нам разрешено двигаться на одну ячейку вверх или на одну ячейку вправо. Нужно определить, какое максимальное число фишек можно собрать по пути в самую верхнюю правую ячейку.

Алла

Эту задачу тоже можно решить с применением динамического программирования?

Тимофей

Да. Только она решается не обычным, а двумерным динамическим программированием, так как теперь сохранять значения нужно не в одномерном массиве, а в двумерной матрице. Ведь по полю можно двигаться в двух направлениях.



На каждом шаге игры мы выбираем между двумя вариантами: пойти вверх или пойти вправо

Алла

Кажется, тут не получится собрать все фишки с поля. Например, если двигаться можно только вверх и вправо, то нельзя взять фишки и из ячейки (1, 0), и из ячейки (0, 2).

Тимофей

Верно. Сделав шаг вверх, мы не сможем потом спуститься. Аналогично, выбрав шаг вправо, мы уже не сможем вернуться в клетки, расположенные левее.

Наивное решение

Гоша

Я знаю, как решить эту задачу без динамического программирования. Надо перебрать все возможные пути, подсчитывая, сколько фишек можно собрать на каждом из них, а потом просто вернуть максимум.

Тимофей

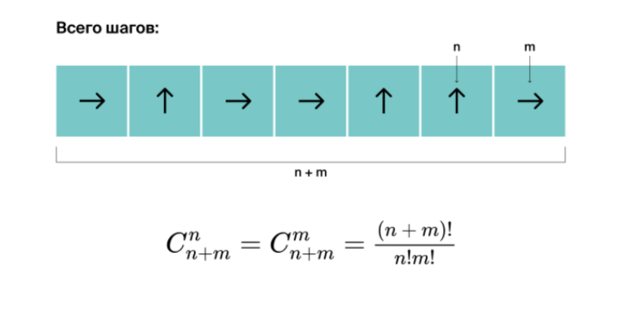
Думаешь, это решение будет эффективным? Можешь посчитать, сколько всего путей придётся рассмотреть?

Чтобы узнать количество различных путей для поля N на M, нам потребуется посчитать «биномиальные коэффициенты». Они записываются как C_n^m и обозначают количество способов выбрать m элементов из n элементов.

Всего на пути из левой нижней ячейки в правую верхнюю проделаем $m + n$ шагов. При этом будет сделано m шагов вправо и n шагов вверх. Путь однозначно задаётся выбором n моментов, когда будет выбрано движение вверх. В остальных m случаях будем двигаться вправо.

То есть количество различных путей будет соответствовать количеству способов выбрать n моментов из $m + n$ вариантов, когда пойти вверх. А это равно C_{n+m}^n.

Кроме этого, ответ можно получить, посчитав количество способов выбрать m перемещений вправо. То есть C_{n+m}^m.

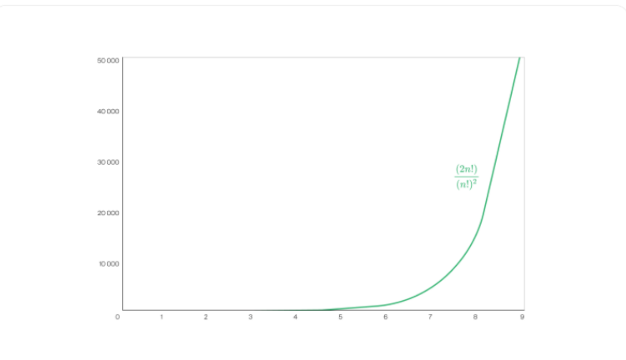


$$C_{n+m}^n = C_{n+m}^m = \frac{(n+m)!}{n!m!}$$

В любом пути по таблице размером N на M будет ровно n шагов вверх и m шагов вправо. Количество путей в таблице вычисляется по формуле биномиальных коэффициентов

Расписав формулу подсчёта биномиальных коэффициентов для этих двух случаев, можно увидеть, что ответ будет одинаковый. Это не совпадение, это свойство биномиальных коэффициентов.

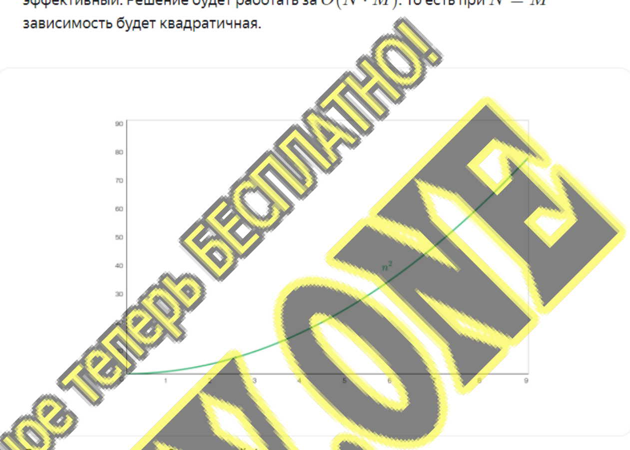
Решение с такой алгоритмической сложностью для больших объёмов данных будет работать очень долго. Так выглядит график функции зависимости времени работы от размера входных данных, если $M = N$:



При n = 8 значение функции 12870, а при n = 9 уже 48620. Скорость роста этой функции очень высокая

Решение с помощью динамики

Вариант с использованием динамического программирования намного более эффективный. Решение будет работать за $O(N \cdot M)$. То есть при $N = M$ зависимость будет квадратичная.



Скорость роста квадратичной функции существенно меньше скорости роста факториальной функции

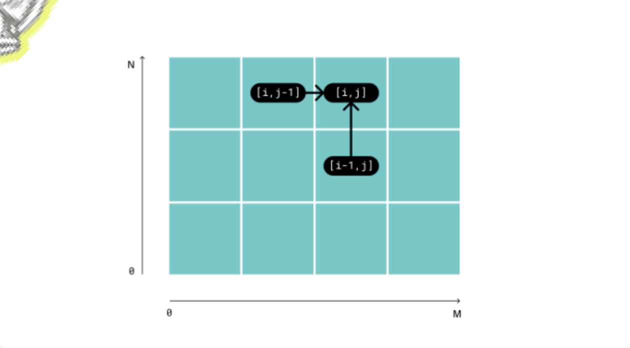
Пусть количество фишек для каждой из ячеек хранится в матрице `points`. Ответим на пять вопросов, которые помогут нам в решении задачи.

Что будет храниться в `dp`? На этот раз `dp` — матрица.

$dp[i][j]$ — максимально возможное число фишек, которое можно собрать, добравшись до ячейки (i, j).

Каким будет базовый случай для задачи? `dp[0][0] = points[0][0]`. То есть на первом шаге мы возьмём столько фишек, сколько есть в стартовой ячейке.

Каким будет переход динамики? В каждую клетку можно прийти либо снизу, то есть из ячейки `dp[i-1][j]`, либо слева, то есть из `dp[i][j-1]`.



В ячейку pdi можно попасть либо из ячейки dpi-1 либо из ячейки dpi

Получим формулу перехода:

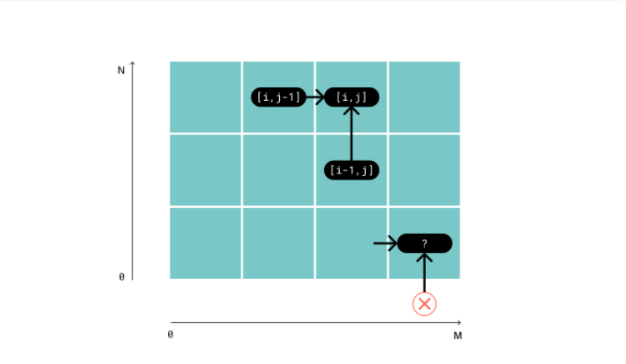
```
dp[i][j] = max(dp[i-1][j], dp[i][j-1]) + points[i][j]
```

Алла

А что делать, если, например, клетки `dp[i-1][j]` нет?

Тимофей

Мне кажется, если какой-то ячейки нет, то можно считать, что в ней записан 0. Так как 0 будет не лучше самого плохого варианта прохода по полю.



Ячейка с индексом по оси i = -1 не существует. Вместо её значения можно использовать 0 или число, означающее отрицательную бесконечность

Каким будет порядок вычисления данных в массиве `dp`? Чтобы посчитать значение в клетке $dp[i, j]$, нужно знать значение в $dp[i-1][j]$ и $dp[i][j-1]$. То есть при вычислении та часть матрицы, которая находится ниже и левее ячейки (i, j), должна быть уже посчитана. Это получится, если обходить матрицу сначала по строкам, потом по столбцам.

Алла

То есть это будет два вложенных цикла?

Тимофей

Да, внешний цикл по возрастанию i, внутренний — по возрастанию j.

Где будет располагаться ответ на исходный вопрос? Искомое значение будет содержаться в ячейке $dp[N-1][M-1]$, то есть в верхней правой клетке матрицы.

Гоша

У меня вопрос по третьему пункту — формуле перехода. Можно ли избежать рассмотрения отдельных случаев, обрабатывающих выход за границы массива?

Тимофей

Можно. Для этого существует приём, который называется «каёмочка».

Алла

Смешное название! И что это за приём?

Тимофей

Его идея вот в чём: добавить один дополнительный ряд снизу и один ряд слева. Значения в добавленных ячейках будут $-\infty$. Можно просто записать большое по модулю отрицательное число. Обходя матрицу, нужно использовать эти ячейки только как вспомогательные при обработке основных ячеек. Тогда выхода за границы массива не будет.

Гоша

Очень интересно! Попробую написать код.

Решите задачу «Поле с цветочками»..

Восстановление ответа

Алла

Допустим, мы получили ответ на задачу. То есть выяснили, какое максимальное количество фишек возможно собрать, пройдя из левой нижней в правую верхнюю клетку. А как понять, какой мы для этого проделали путь?

Тимофей

Путь можно восстановить. По сути, для этого нужно решить обратную задачу.

Допустим, мы находимся в правой верхней клетке. То есть в ячейке с индексами $dp[n-1][m-1]$. Попасть туда мы могли двумя способами: из ячейки $dp[n-2][m-1]$ или из $dp[n-1][m-2]$.

Чтобы понять, каким из двух путей мы пришли, нужно сравнить значения в этих двух ячейках. Когда мы считали значения в массиве dp мы пользовались формулой

$dp[i][j] = dp[x][y] + points[i][j]$.

При этом $dp[x][y]$ это максимальное значение из $dp[i-1][j]$ и $dp[i][j-1]$. Следовательно, ячейка из которой мы пришли имеет большее значение. Если значение в клетке снизу больше, то путь проходил через неё. Иначе — через ячейку слева от нас. Выбранное направление записываем в массив, чтобы потом восстановить оптимальный маршрут.

Перейдём в выбранную ячейку и повторим операцию. И так до тех пор, пока не окажемся в нижней левой клетке. После этого массив, который соответствует полученному пути, нужно развернуть, ведь он записан с конца.

Нужно принимать во внимание случай, когда значения в ячейках снизу и слева равны. Это значит, что мы могли прийти из любой из этих клеток, следовательно, оптимальных решений может быть несколько, и для каждого из них будет свой путь. Но количество собранных фишек для каждого из таких решений будет одинаковым.

Решите задачу «Сложное поле с цветочками».

Вперёд

Задача о рюкзаке. Динамическое решение

Тимофей
Вы ещё не забыли про задачу о моём рюкзаке с провиантом?

Гоша
Но ведь мы её уже решили.

Тимофей
Решили, верно. Но время выполнения вашего алгоритма $O(2^n)$ — это слишком долго. Давайте расскажу, как с помощью динамического программирования получить более быстрое решение.

Разбиваем задачу на подзадачи

Как мы уже говорили, динамическое программирование позволяет решить задачу, если мы найдём ответы на её подзадачи и придумаем, как их объединить. В ситуации с рюкзаком Тимофея разбивка на подзадачи нетривиальная и от этого особенно увлекательная.

Напомним формулировку: есть рюкзак вместимостью M, а также N различных предметов. У каждого предмета есть ценность $cost_i$ (например, для рюкзака Тимофея ценностью является калорийность продукта) и масса $weight_i$. При этом каждый предмет можно либо взять целиком, либо не брать вовсе. Взять часть предмета невозможно. Надо собрать такой набор предметов, что их суммарная масса не больше M, а суммарная ценность максимальная.

Одним из параметров динамики будет вместимость рюкзака. Если мы знаем, какую наибольшую суммарную стоимость можно получить для рюкзака вместимостью j, для всех целых $j < M$, то мы можем перебрать все имеющиеся у нас предметы и узнать, какую наибольшую суммарную стоимость можно получить для рюкзака вместимостью M.

К примеру, мы знаем ответы для подзадач: лучшая ценность рюкзака вместимостью 1 кг — 400 ккал, вместимостью 2 кг — 490 ккал. Мы хотим узнать наибольшую ценность для рюкзака вместимостью 3 кг, если у нас есть ещё пара предметов, которые можно добавить: банка горошка весом 2 кг и ценностью 100 ккал и коробка чая весом 1 кг и ценностью 90 ккал. Для того чтобы получить ответ для подзадачи весом 3 кг, мы попробуем четыре разных варианта:

1. К рюкзаку весом 1 кг добавим 1 кг чая, получим калорийность 400 + 90 = 490 ккал. Этот вариант можно было не рассматривать, так как мы заранее знали максимальную ценность для рюкзака весом 2 кг. Никакая другая комбинация предметов не позволит получить ценность выше, чем 490 ккал. В противном случае ответ для подзадачи весом 2 кг некорректный.
2. Возьмём 2 кг горошка и добавим к нему лучшее решение для подзадачи вместимостью `3 кг` — `2 кг` = 1 кг. Лучшее решение для рюкзака вместимостью 1 кг — 400 ккал. Итог: 100 + 400 = 500 ккал.
3. Возьмём 1 кг чая и к нему добавим лучшее решение для подзадачи вместимостью `3 кг` — `1 кг` = 2 кг. Лучшее решение для рюкзака вместимостью 2 кг — 490 ккал. Итог: 90 + 490 = 580 ккал.
4. К рюкзаку весом 2 кг попробуем добавить 2 кг горошка, но не сможем. Так как вместимость рюкзака всего 3 кг.

Но что если идеальное наполнение рюкзака вместимостью 2 кг — это упаковка чая весом 1 кг + содержимое рюкзака вместимостью 1 кг? В этом случае самый выгодный вариант с калорийностью 580 ккал будет невалидным, так как пачку чая мы уже использовали, и второй такой же у нас нет. Чтобы случайно не попытаться положить один и тот же предмет в рюкзак дважды, вторым параметром динамики будет множество предметов, которые мы всё ещё можем положить в рюкзак.

Гоша
Подожди, я немного запутался. Давай вернёмся к алгоритму решения задач динамического программирования и рассмотрим по нему.

Помню, что первый пункт — определить, что будем хранить в `dp` — одномерном или двумерном массиве динамики.

Тимофей
Это зависит от того, на какой вопрос ты хочешь получить ответ. Давай пока вспомним мой рюкзак для похода и посчитаем, какую наилучшую суммарную калорийность я могу набрать.

Разбор задачи по алгоритму

Для удобства повествования пронумеруем все пачки с крупами числами от 0 до $N - 1$. Гречневая крупа будет идти под номером 0, рисовая — 1, пшённая — 2, манная — 3.

Что будет храниться в `dp`? В этом случае `dp[i][j]` — наибольшая калорийность, которую мы сможем получить, если будем считать, что у Тимофея есть только крупы с номерами от 0 до i и рюкзак вместимостью j.

Каким будет базовый случай для задачи? Базовых случаев такой динамики N — ровно столько же, сколько у Тимофея есть разных упаковок круп. Если вместимость рюкзака 0, то вне зависимости от набора круп, которые есть у Тимофея в кладовке, суммарная калорийность будет 0. Кроме того, если круп нет совсем, то любой рюкзак окажется пуст и его суммарная калорийность будет равна 0.

Каким будет переход динамики? Переход динамики мы рассмотрели чуть выше. Нам надо взять максимум из двух величин:

1. Наилучшая калорийность рюкзака вместимости j, если мы не берём i-й предмет. В этом случае наилучшая калорийность будет такой же, как если мы рассматриваем первые $i - 1$ круп и рюкзак вместимостью j: $dp[i - 1][j]$.
2. Сумма калорийности i-й крупы, которую мы кладём в рюкзак, и наилучшей калорийности рюкзака, размер которого соответствует оставшемуся свободному месту в нашем рюкзаке: $cost[i] + dp[i - 1][j - weight[i]]$. При этом если $j - weight[i] \leq 0$, то калорийность такого рюкзака должна приравниваться к нулю, так как формально мы получили рюкзак отрицательной вместимости, то есть ни один предмет не может поместиться в такой рюкзак.

Итоговая формула перехода выглядит так

```
dp[i][j] = max(dp[i - 1][j], cost[i] + dp[i - 1][j - weight[i]])
```

Каким будет порядок вычисления данных в массиве `dp`? Заполняем сначала целиком первую строку матрицы, потом вторую и так далее — пока не заполним i-ю строку.

Давайте рассмотрим чуть подробнее первую строку. Первая крупа имеет вес 4 кг и энергетическую ценность 13 200 ккал. Тогда для рюкзаков вместимостью меньше 4 кг максимальная ценность рюкзака будет 0, так как при попытке положить эту пачку крупы в рюкзак возникнет перевес. Для всех рюкзаков вместимостью больше либо равной 4 кг максимальная ценность 13 200 ккал. Чем больше будет рюкзак, тем больше свободного места останется, но доложить в него на первой строке больше нечего, так как у нас на рассмотрении всего 1 пачка крупы.

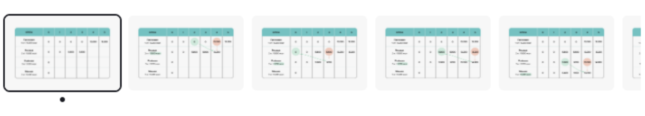
КРУПА	0	1	2	3	4	5
Гречневая 4 кг / 13.200 ккал	0	0	0	0	13.200	13.200
Рисовая 2 кг / 5.800 ккал	0					
Пшённая 3 кг / 9.150 ккал	0					
Манная 2 кг / 6.400 ккал	0					

Где будет располагаться ответ на исходный вопрос? В изначальной формулировке предполагается, что у нас есть N предметов и рюкзак вместимостью M, следовательно, ответ будет лежать в правом нижнем углу — в ячейке `dp[N][M]`.

Вот как будет заполняться таблица `dp`:

КРУПА	0	1	2	3	4	5
Гречневая 4 кг / 13.200 ккал	0	0	0	0	13.200	13.200
Рисовая 2 кг / 5.800 ккал	0	0	5.800	5.800		
Пшённая 3 кг / 9.150 ккал	0					
Манная 2 кг / 6.400 ккал	0					

Аналогично заполняем вторую строку. Для рюкзаков вместимостью меньше 4 мы даже не пытаемся класть гречку, так как её вес больше вместимости рюкзака. Зато рисовую крупу мы можем положить уже в рюкзак размера 2



Какая пространственная сложность этого алгоритма?

- $O(N^2)$
- $O(M^2)$
- $O(N \cdot M)$
- $O(N + M)$
- $O(M \log N)$
- $O(N \log M)$

Пространственная сложность $O(N \cdot M)$, так как мы должны хранить массив `dp`, состоящий из N строк и M столбцов.

Пространственную сложность этого алгоритма можно улучшить до $O(M)$. Для вычисления рекуррентной формулы мы всегда смотрим только на предыдущую строчку таблицы и никогда не смотрим на более ранние строки, а значит их можно не хранить.

Какая алгоритмическая сложность этого алгоритма?

- $O(N^2)$
- $O(M^2)$
- $O(N + M)$
- $O(N \cdot M)$
- $O(M \log N)$
- $O(N \log M)$

Алгоритмическая сложность $O(N \cdot M)$, так как мы должны обновить каждую из ячеек массива `dp`, состоящего из N строк и M столбцов. Каждое обновление происходит за константное время.

Восстановление набора продуктов

Гоша
Мы узнали, какая максимальная калорийность может быть у твоего рюкзака. Благодаря этому можно понять, сколько дней ты можешь провести в походе: для этого достаточно разделить суммарную калорийность на твою суточную норму калорий. Но как узнать, что именно требуется положить в рюкзак?

Тимофей
Для этого давай восстановим список вещей для рюкзака.

Чтобы удобно восстанавливать ответ, в массиве `dp` можно хранить не только наилучшую возможную калорийность, но и список круп, благодаря которым эту пищевую ценность можно получить.

Кроме изменения того, что хранится в массиве, изменится и алгоритм перехода.

- Если мы не берём i-й предмет, то мы берём список круп, актуальный для рюкзака такой же вместимости j, но собранного из первых $i - 1$ круп.
- Если мы берём i-ю крупу, то мы берём список круп, актуальный для рюкзака вместимости $j - weight[i]$, собранного из первых $i - 1$ круп.

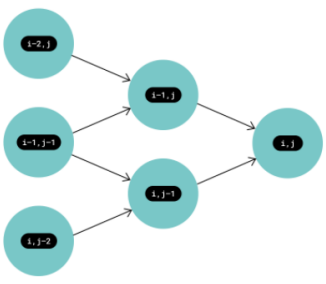
Гоша
Для того чтобы хранить список провизии, позволяющий собрать самый сытный рюкзак, нам придётся задействовать много дополнительного места в памяти. Можно ли как-то этого избежать?

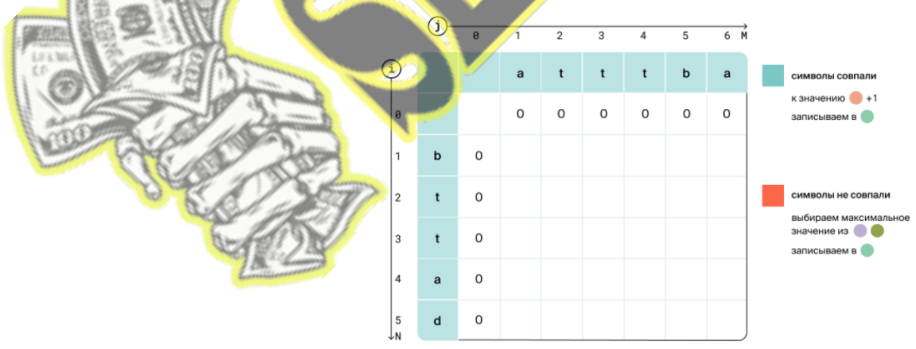
Тимофей
Да, можно. Но только если мы целиком храним массив `dp`. Тогда, как и в задаче с полем и фишками, мы всякий раз сможем вычислять, какой предмет мы положили, а какой нет, опираясь на информацию о предмете и на ответы для подзадач.

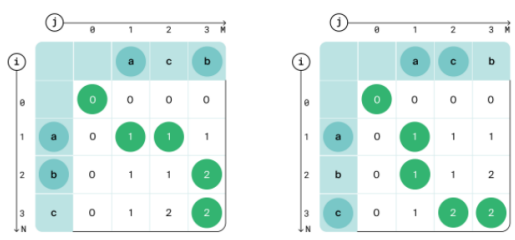
Мы рассмотрели задачу о рюкзаке. При этом мы использовали целочисленные значения весов и вместимости, чтобы удобно перебирать индексы в массиве dp. Однако если в задаче присутствуют произвольные дробные веса, то её невозможно решить ни жадным алгоритмом, ни алгоритмом динамического программирования.

Решите задачу «Золото лепреконов».

Вперёд







Я Практикум
- Жадные алгоритмы и динамическое программирование
- Жадные алгоритмы
- Задача о расписании
- Задача о рюкзаке
- Динамическое программирование. М...
- Динамическое программирование. А...
- Двумерная динамика
- Задача о рюкзаке. Динамическое реш...
- Наибольшая общая подпоследовател...
- Задача о рюкзаке. Динамическое реш...
- Наибольшая общая подпоследовател...
- Задача о рюкзаке. Динамическое реш...
- Наибольшая общая подпоследовател...
- Разбор задачи: переход от теории к п...
- Резюме
- Задача о рюкзаке. Динамическое реш...
- Наибольшая общая подпоследовател...
- Разбор задачи: переход от теории к п...
- Резюме
- Задача о рюкзаке. Динамическое реш...
- Наибольшая общая подпоследовател...
- Разбор задачи: переход от теории к п...
- Резюме
- Задача о рюкзаке. Динамическое реш...
- Наибольшая общая подпоследовател...
- Разбор задачи: переход от теории к п...
- Резюме
- Задача о рюкзаке. Динамическое реш...
- Наибольшая общая подпоследовател...
- Разбор задачи: переход от теории к п...
- Резюме
- Задача о рюкзаке. Динамическое реш...
- Наибольшая общая подпоследовател...
- Разбор задачи: переход от теории к п...
- Резюме
- Задача о рюкзаке. Динамическое реш...
- Наибольшая общая подпоследовател...
- Разбор задачи: переход от теории к п...
- Резюме
- Задача о рюкзаке. Динамическое реш...
- Наибольшая общая подпоследовател...
- Разбор задачи: переход от теории к п...
- Резюме
- Задача о рюкзаке. Динамическое реш...
- Наибольшая общая подпоследовател...
- Разбор задачи: переход от теории к п...
- Резюме
- Задача о рюкзаке. Динамическое реш...
- Наибольшая общая подпоследовател...
- Разбор задачи: переход от теории к п...
- Резюме
- Задача о рюкзаке. Динамическое реш...
- Наибольшая общая подпоследовател...
- Разбор задачи: переход от теории к п...
- Резюме
- Задача о рюкзаке. Динамическое реш...
- Наибольшая общая подпоследовател...
- Разбор задачи: переход от теории к п...
- Резюме
- Задача о рюкзаке. Динамическое реш...
- Наибольшая общая подпоследовател...
- Разбор задачи: переход от теории к п...
- Резюме
- Задача о рюкзаке. Динамическое реш...
- Наибольшая общая подпоследовател...
- Разбор задачи: переход от теории к п...
- Резюме
- Задача о рюкзаке. Динамическое реш...
- Наибольшая общая подпоследовател...
- Разбор задачи: переход от теории к п...
- Резюме
- Задача о рюкзаке. Динамическое реш...
- Наибольшая общая подпоследовател...
- Разбор задачи: переход от теории к п...
- Резюме
- Задача о рюкзаке. Динамическое реш...
- Наибольшая общая подпоследовател...
- Разбор задачи: переход от теории к п...
- Резюме
- Задача о рюкзаке. Динамическое реш...
- Наибольшая общая подпоследовател...
- Разбор задачи: переход от теории к п...
- Резюме
- Задача о рюкзаке. Динамическое реш...
- Наибольшая общая подпоследовател...
- Разбор задачи: переход от теории к п...
- [illegible]
- Задача о рюкзаке. Динамическое реш...
- Наибольшая общая подпоследовател...
- Разбор задачи: переход от теории к п...
- Резюме
- Оцените пройденные уроки
- Финальное задание спринта 7

СПРИНТ 8
- Алгоритмы на строках
- Финальное задание спринта 8
- Завершение курса

Наибольшая общая подпоследовательность

Поиск `наибольшей общей подпоследовательности` (сокращённо НОП) часто встречается в разных задачах. С его помощью можно проверять работы студентов на списывание, искать опечатки в текстах и красиво отображать различия в файлах для код-ревью.

> Подпоследовательность — это не то же самое, что подстрока. Элементы подпоследовательности не обязаны располагаться подряд. Однако они должны сохранить свой исходный порядок друг относительно друга.

В этом уроке мы разберём, как решать задачу поиска наибольшей общей подпоследовательности с помощью двумерной динамики.

Даны две последовательности: $a_1, a_2, ...a_n$ и $b_1, b_2, ..., b_m$. Нужно найти их самую длинную общую подпоследовательность.

Например, для последовательностей $\{1, 3, 2, 3, 5, 7\}$ и $\{5, 1, 2, 3, 5, 9, 7\}$ ответом будет $\{1, 2, 3, 5, 7\}$.

Рекурсивное решение

Попробуем решать эту задачу с конца. Рассмотрим самые правые символы в каждой строке: a_n и b_m соответственно.

- Если они равны, значит НОП имеет длину как минимум 1. Отбрасываем последние символы в каждой строке и повторяем рассмотрение для двух получившихся после отбрасывания строк.
- Если они не равны, то надо рассмотреть две подзадачи. В первой подзадаче мы не берём последний символ строки b и ищем НОП строк $a_1, ..., a_n$ и $b_1, ...b_{m-1}$, а во второй — не берём последний символ строки a и ищем НОП для строк $a_1, ..., a_{n-1}$ и $b_1, ..., b_m$.

Так как мы ищем наибольшую общую подпоследовательность, то из результатов двух подзадач мы выберем максимальный.

Длины наибольших общих подпоследовательностей считаются рекурсивно. Сначала у нас 2 варианта откинуть суммарно 1 символ из двух строк, потом 3 варианта откинуть 2 символа из двух строк и так далее

Мы получили рекурсивное решение, которое не сложно развернуть и записать в виде алгоритма динамического программирования.

Решение с помощью динамики

Что будет храниться в `dp`? В $dp[i][j]$ будем хранить длину НОП для подстрок $a[1:i]$ и $b[1:j]$. Такие подстроки называются `префиксами`. Будьте внимательны, нумерация индексов в непустых строках начинается не с 0, а с 1.

Каким будет базовый случай для задачи? Как мы рассмотрели выше, если какая-то из строк пустая, то есть $i = 0$ или $j = 0$, то их НОП равна 0: $dp[i][j] = 0$.

Каким будет переход динамики? В процессе перехода мы удлиняем одну из строк на один символ.

- Если до удлинения крайние символы в двух строках были равны и включены в НОП, то добавление нового символа не изменит длину НОП.
- Иначе надо сравнить, не совпал ли новый символ с символом на конце второй строки. Если совпал, то увеличиваем НОП.

Каким будет порядок вычисления данных в массиве `dp`? Будем сначала считать, что строка a состоит из одного символа, а строка b постепенно удлиняется. Потом удлиняем строку a и снова перебираем все возможные длины b.

Где будет располагаться ответ на исходный вопрос? Длина наибольшей общей подпоследовательности будет находиться в ячейке $dp[N][M]$.

Рассмотрим работу этого алгоритма на примере строк $A = \{b, t, t, a, d\}$ и $B = \{a, t, t, t, b, a\}$.

i \ j		0	1	2	3	4	5	6
			a	t	t	t	b	a
0			0	0	0	0	0	0
1	b	0						
2	t	0						
3	t	0						
4	a	0						
5	d	0						

Зададим базовый случай: если одна из подстрок пустая, то длина НОП двух подстрок равна 0

Восстановление ответа

Для того чтобы по таблице длин НОП получить наибольшую общую подпоследовательность, необходимо выполнить следующие действия:

1. Завести массив `answer`, в котором будет храниться НОП, записанная от конца к началу.
2. Начать с клетки $dp[N][M]$. Приравнять i к N, j к M.
3. Если $A[i] = B[j]$, нужно записать в `answer` символ $A[i]$ и переместиться в клетку $dp[i-1][j-1]$. Как мы говорили в начале урока, рассмотренный символ точно является частью НОП.
4. Если $A[i] \neq B[j]$, то один из символов в строках точно не входит в НОП.

- Если $dp[i][j] = dp[i-1][j]$, значит, существует НОП, в которую $A[i]$ точно не входит. Перемещаемся вверх — в соседнюю ячейку.
- Иначе — если $dp[i][j] = dp[i][j-1]$, значит, существует НОП, в которую $B[j]$ не входит. Перемещаемся в ячейку, расположенную левее текущей.

i \ j		0	1	2	3
			a	c	b
0		0	0	0	0
1	a	0	1	1	1
2	b	0	1	1	2
3	c	0	1	2	2

i \ j		0	1	2	3
			a	c	b
0		0	0	0	0
1	a	0	1	1	1
2	b	0	1	1	2
3	c	0	1	2	2

Для строк abc и acb существует две НОП. В зависимости от того какое условие перехода приоритетнее(шаг вверх или шаг влево) мы получим либо НОП = ab, либо — ac

5. Повторяем пункты 3–4 до тех пор, пока $dp[i][j] \neq 0$.
6. Переворачиваем массив `answer` и получаем наибольшую общую подпоследовательность.

Сложность алгоритма

В процессе заполнения мы обошли $M \cdot N$ клеток и в каждой из них вычислили результат за константное время. Следовательно, сложность построения таблицы `dp` равна $O(N \cdot M)$.

Кроме того, для восстановления ответа мы сделали не более N шагов вверх по таблице и не более M шагов влево. Итого сложность восстановления НОП по массиву `dp` равна $O(N + M)$.

Решите задачу «Гороскопы».

Вперёд

Жадные алгоритмы и динамическое программирование
Жадные алгоритмы
Задача о расписании
Задача о рюкзаке
Динамическое программирование. М...
Динамическое программирование. А...
Двумерная динамика
Задача о рюкзаке. Динамическое реш...
Наибольшая общая подпоследовател...
Задача о рюкзаке. Динамическое реш...
Наибольшая общая подпоследовател...
Задача о рюкзаке. Динамическое реш...
Наибольшая общая подпоследовател...
Разбор задачи: переход от теории к п...
Резюме
Задача о рюкзаке. Динамическое реш...
Наибольшая общая подпоследовател...
Разбор задачи: переход от теории к п...
Резюме
Задача о рюкзаке. Динамическое реш...
Наибольшая общая подпоследовател...
Разбор задачи: переход от теории к п...
Резюме
Задача о рюкзаке. Динамическое реш...
Наибольшая общая подпоследовател...
Разбор задачи: переход от теории к п...
Резюме
Задача о рюкзаке. Динамическое реш...
Наибольшая общая подпоследовател...
Разбор задачи: переход от теории к п...
Резюме
Задача о рюкзаке. Динамическое реш...
Наибольшая общая подпоследовател...
Разбор задачи: переход от теории к п...
Резюме
Задача о рюкзаке. Динамическое реш...
Наибольшая общая подпоследовател...
Разбор задачи: переход от теории к п...
Резюме
Задача о рюкзаке. Динамическое реш...
Наибольшая общая подпоследовател...
Разбор задачи: переход от теории к п...
Резюме
Задача о рюкзаке. Динамическое реш...
Наибольшая общая подпоследовател...
Разбор задачи: переход от теории к п...
Резюме
Задача о рюкзаке. Динамическое реш...
Наибольшая общая подпоследовател...
Разбор задачи: переход от теории к п...
Резюме
Задача о рюкзаке. Динамическое реш...
Наибольшая общая подпоследовател...
Разбор задачи: переход от теории к п...
Резюме
Задача о рюкзаке. Динамическое реш...
Наибольшая общая подпоследовател...
Разбор задачи: переход от теории к п...
Резюме
Задача о рюкзаке. Динамическое реш...
Наибольшая общая подпоследовател...
Разбор задачи: переход от теории к п...
Резюме
Задача о рюкзаке. Динамическое реш...
Наибольшая общая подпоследовател...
Разбор задачи: переход от теории к п...
Резюме
Задача о рюкзаке. Динамическое реш...
Наибольшая общая подпоследовател...
Разбор задачи: переход от теории к п...
Резюме
Задача о рюкзаке. Динамическое реш...
Наибольшая общая подпоследовател...
Разбор задачи: переход от теории к п...
Резюме
Задача о рюкзаке. Динамическое реш...
Наибольшая общая подпоследовател...
Разбор задачи: переход от теории к п...
Резюме
Задача о рюкзаке. Динамическое реш...
Наибольшая общая подпоследовател...
Разбор задачи: переход от теории к п...
Резюме
Задача о рюкзаке. Динамическое реш...
Наибольшая общая подпоследовател...
Разбор задачи: переход от теории к п...
Резюме
Задача о рюкзаке. Динамическое реш...
Наибольшая общая подпоследовател...
Разбор задачи: переход от теории к п...
Резюме
Задача о рюкзаке. Динамическое реш...
Наибольшая общая подпоследовател...
Разбор задачи: переход от теории к п...
Резюме
Задача о рюкзаке. Динамическое реш...
Наибольшая общая подпоследовател...
Разбор задачи: переход от теории к п...
Резюме
Задача о рюкзаке. Динамическое реш...
Наибольшая общая подпоследовател...
Разбор задачи: переход от теории к п...
Резюме
Задача о рюкзаке. Динамическое реш...
Наибольшая общая подпоследовател...
Разбор задачи: переход от теории к п...
Резюме
Задача о рюкзаке. Динамическое реш...
Наибольшая общая подпоследовател...
Разбор задачи: переход от теории к п...
Резюме
Задача о рюкзаке. Динамическое реш...
Наибольшая общая подпоследовател...
Разбор задачи: переход от теории к п...
Резюме
Задача о рюкзаке. Динамическое реш...
Наибольшая общая подпоследовател...
Разбор задачи: переход от теории к п...
Задача о рюкзаке. Динамическое реш...
Наибольшая общая подпоследовател...
Разбор задачи: переход от теории к п...
Резюме
Оцените пройденные уроки
Финальное задание спринта 7
СПРИНТ 8
Алгоритмы на строках
Финальное задание спринта 8
Завершение курса

Разбор задачи: переход от теории к практике

В одном из предыдущих уроков мы рассмотрели классическую задачу динамического программирования о нахождении наибольшей общей подпоследовательности. Теперь давайте разберём похожую задачу — нахождение наибольшей возрастающей подпоследовательности (далее будем использовать сокращённый вариант названия — НВП).

В данном случае мы будем работать с одной последовательностью, а не с двумя, и будем обозначать её как последовательность A длины n.

Наибольшая возрастающая подпоследовательность (НВП)

Можно переформулировать задачу следующим образом: из последовательности A необходимо вычеркнуть наименьшее число элементов так, чтобы оставшиеся числа шли строго по возрастанию.

Рассмотрим несколько примеров.

Пример 1:

$A = \{1, 2, 3, 4, 5, 6\}$. НВП совпадает со всей последовательностью, её длина равна 6.

Пример 2:

$A = \{6, 5, 4, 3, 2, 1\}$. Здесь какие бы два элемента мы ни взяли, они будут идти в порядке убывания, значит, длина НВП равна 1.

> Заметим, что в любой непустой последовательности найдётся возрастающая подпоследовательность из одного элемента, так как отдельное число тоже является подпоследовательностью.

Пример 3:

$A = \{4, 1, 10, 0, 8, 10, 2, 3, 9\}$. Найдём некоторые возрастающие подпоследовательности: например, $\{4, 8, 10\}$, $\{1, 2, 3\}$ и $\{0, 2, 3, 9\}$. Длина НВП в данном случае составляет 4. Сама НВП определяется неоднозначно — мы можем взять в качестве ответа как $\{0, 2, 3, 9\}$, так и $\{1, 2, 3, 9\}$.

Найдите длину НВП данной последовательности:

$A = \{5, 6, 2, 3, 7, 8, 10, 1, 4, 11, 5, 6\}$

○ 1
○ 2
○ 3
○ 4
○ 5
● 6

Верно! Один из вариантов НВП длины 6 — $\{5, 6, 7, 8, 10, 11\}$, но есть и вариант $\{2, 3, 7, 8, 10, 11\}$.

Дальнейшая теория даст вам идеи для решения задачи «Путешествие».

Наибольшая общая возрастающая подпоследовательность — наивное решение

Усложним задачу: для данных последовательностей A и B требуется найти наибольшую общую возрастающую подпоследовательность (далее — НОВП). Рассмотрим пример. Даны две последовательности:

$A = \{3, 8, 9, 1, 3, 1, 2, 1\}$

$B = \{1, 3, 1, 2, 8, 9, 1\}$

Их наибольшая общая подпоследовательность имеет длину 5 — это $\{1, 3, 1, 2, 1\}$. Попробуем выделить в общей подпоследовательности наибольшую возрастающую подпоследовательность — получим $\{1, 2\}$ или $\{1, 3\}$. Однако в A и B есть общая возрастающая подпоследовательность большей длины — это $\{3, 8, 9\}$. Таким образом, найти сначала НОП, а потом выделить в ней НВП не получится. Нужна новая динамика.

Рассмотрим наивное решение, которое для последовательностей A и B длины n и m соответственно будет работать за $O(n^2 \cdot m^2)$, а затем улучшим его.

Пусть $dp[i][j]$ — это длина НОВП на префиксах (префикс последовательности A длины L — это её начало $A[0 : L - 1]$) длины i (в A) и j (в B), причём элементы на позициях i и j обязательно входят в НОВП. Из этого определения сразу следует, что $A[i - 1] == B[j - 1]$. Научимся вычислять данное `состояние` динамики.

Во-первых, если $A[i - 1] \neq B[j - 1]$, то $dp[i][j] = 0$. В ином случае необходимо перебрать предыдущие элементы в последовательностях A и B.

```python
# Заполняем массив нулями, так как длина НОВП в худшем случае равна нулю.
# Бо́льшую оценку без просмотра последовательностей мы не можем получить.
dp = [[0] for i in range(n + 1)] for j in range(m + 1)]
for i in range(1, n + 1):
  for j in range(1, m + 1):
    if A[i - 1] == B[j - 1]:
      max_len = 0
      for i_prev in range(1, i):
        if A[i_prev - 1] >= A[i - 1]:
          continue
        for j_prev in range(1, j):
          max_len = max(max_len, dp[i_prev][j_prev])
      dp[i][j] = max_len + 1
# Надо перебрать все пары концов НОВП, чтобы найти оптимальный ответ.
answer = 0
for i in range(n + 1):
  for j in range(m + 1):
    answer = max(answer, dp[i][j])
```

Первая оптимизация

Попробуем переформулировать состояние динамики, чтобы можно было делать переход более эффективно.

Пусть $dp[i][j]$ — также длина НОВП на префиксах длины i и j соответственно, но элемент $A[i - 1]$ теперь не обязан входить в НОВП. $B[j - 1]$ по-прежнему входит в НОВП, парный ему элемент в A лежит где-то на префиксе последовательности длины i.

Рассмотрим два варианта. Если $A[i - 1]$ входит в НОВП, то обязательно $A[i - 1] == B[j - 1]$. В этом случае надо будет перебрать только предыдущий элемент последовательности B, это обеспечит нам возрастание получающейся при переходах последовательности. В A не надо перебирать последний элемент, достаточно обращаться в первом индексе к $i - 1$. По определению, последний элемент НОВП будет лежать где-то между элементами 0 и $i - 2$ включительно (при индексации с нуля), что позволяет корректно обновить $dp[i][j]$.

Если же $A[i - 1]$ не входит в НОВП, то попробуем обновить ответ, проигнорировав $A[i - 1]$. Для этого обратимся к $dp[i - 1][j]$.

```python
dp = [[0] for i in range(n + 1)] for j in range(m + 1)]
for i in range(1, n + 1):
  for j in range(1, m + 1):
    # Сначала обновляем состояние согласно
    # второму варианту, когда A[i-1] не входит в НОВП.
    dp[i][j] = dp[i - 1][j]
    # Если A[i - 1] == B[j - 1], то можно сделать обновление dp[i][j] по первому
    # варианту, когда A[i - 1] включается в НОВП.
    if A[i - 1] == B[j - 1]:
      max_val = 0
      for j_prev in range(1, j):
        if B[j_prev - 1] < B[j - 1]:
          max_val = max(max_val, dp[i - 1][j_prev])
        dp[i][j] = max(dp[i][j], 1 + max_val)
# Ответ находится в последней строке таблицы,
# но необязательно в клетке dp[n][m].
ans = 0
for j in range(m + 1):
  ans = max(ans, dp[n][j])
```

Сложность этой версии решения — $O(n \cdot m^2)$, поэтому оптимально выбирать в качестве A более длинную последовательность, а в качестве B — более короткую.

Вторая оптимизация

Сделаем финальную оптимизацию. Зафиксируем i и посмотрим, что происходит в двух вложенных циклах. Чем отличаются итерации цикла по `j_prev` в момент j и $j + 1$?

цикл до j

```python
for j_prev in range(1, j):
  if B[j_prev - 1] < B[j - 1]:
    max_val = max(max_val, dp[i - 1][j_prev])
```

цикл до j+1

```python
for j_prev in range(1, j + 1):
  if B[j_prev - 1] < B[j]:
    max_val = max(max_val, dp[i - 1][j_prev])
```

Теперь заметим, что при фиксированном i мы зайдём в данный цикл, только если $A[i - 1] == B[j - 1]$ или $A[i - 1] == B[j]$. Это значит, что можно переписать циклы следующим образом:

цикл до j

```python
for j_prev in range(1, j):
  if B[j_prev - 1] < A[i - 1]:
    max_val = max(max_val, dp[i - 1][j_prev])
```

цикл до j+1

```python
for j_prev in range(1, j + 1):
  if B[j_prev - 1] < A[i - 1]:
    max_val = max(max_val, dp[i - 1][j_prev])
```

Мы делаем одни и те же шаги при j и при $j + 1$, но при $j + 1$ продвигаемся на единицу дальше при подсчёте максимума. Значит, можно накапливать текущий максимум `dp[i - 1][j_prev]` по всем `j_prev` таким, что `B[j_prev - 1] < A[i - 1]`. Это избавит нас от третьего цикла и сведёт асимптотику к $O(n \cdot m)$.

Так как мы не меняли само состояние динамики, то ответ высчитывается так же, как и в решении за $O(n \cdot m^2)$.

```python
dp = [[0] for i in range(n + 1)] for j in range(m + 1)]
for i in range(1, n + 1):
  max_val = 0
  for j in range(1, m + 1):
    dp[i][j] = dp[i - 1][j]
    if A[i - 1] == B[j - 1]:
      dp[i][j] = max(dp[i][j], max_val + 1)
    if B[j - 1] < A[i - 1]:
      max_val = max(max_val, dp[i - 1][j])
```

Итак, мы подробно разобрали последовательное решение задачи о наибольшей общей возрастающей подпоследовательности. Оттолкнувшись от наивного подхода, мы улучшили наше решение с асимптотики $O(n^2 \cdot m^2)$ до $O(n \cdot m)$ с помощью пары интересных оптимизаций.

Вперёд

Резюме

В этом спринте мы говорили о задачах оптимизации, которые в общем случае сводятся к поиску экстремума — то есть наибольшего или, наоборот, наименьшего значения функции на заданном множестве.

Мы разобрали два подхода, которые применяются для решения экстремальных задач: жадные алгоритмы и динамическое программирование.

Жадные алгоритмы работают быстро. На каждом шаге они выбирают локально оптимальное решение, а последствия этого выбора и прошлые результаты из рассмотрения исключаются. Однако оптимальное тактическое решение не всегда совпадает с лучшим стратегическим решением.

Существуют задачи, в которых невозможно найти точное жадное решение. В таких случаях жадные алгоритмы применяются для нахождения приближённых решений. Приближённое решение может существенно отличаться от точного, зато оно будет посчитано быстро.

Проверить, даст ли алгоритм точное решение или только приближённое, можно с помощью методов формальных доказательств. Мы рассмотрели четыре таких метода:

- Метод полного перебора. Для того чтобы доказать утверждение данным способом, необходимо перебрать все значения, к которым это утверждение когда-либо может быть применено, и проверить для них его корректность.
- Метод прямого следования. Доказательство методом прямого следования подразумевает переход от изначально верного утверждения к тому, которое мы доказываем, при помощи аксиом и уже доказанных утверждений.
- Метод доказательства от противного. В этом методе исходное предположение считается неверным. И отталкиваясь от этого мы пытаемся доказать другое утверждение методом прямого следования — до тех пор, пока не столкнёмся с противоречием.
- Метод математической индукции. В методе математической индукции какое-то утверждение сначала проверяют для базового случая, а потом доказывают, что оно не перестаёт быть корректным при переходе к данным большего размера.

Метод динамического программирования позволяет разработчику гарантированно найти точное решение экстремальной задачи, которая разбивается на несколько подзадач. После того как все подзадачи решены, их ответы необходимо объединить — для вычисления ответа на исходную задачу.

Для составления алгоритма необходимо ответить на 5 вопросов:

1. Что будет храниться в массиве `dp`?
2. Каким будет базовый случай для этой задачи?
3. Каким будет переход динамики? Переход динамики также называют рекуррентной формулой.
4. Каким будет порядок вычисления данных в массиве `dp`?
5. Где будет располагаться ответ на исходный вопрос?

Мы рассмотрели одномерную и двумерную динамики. А ещё разобрали, как решить классическую задачу о рюкзаке, используя двумерную динамику.

Также мы подробно рассмотрели решения двух популярных задач: поиск наибольшей общей подпоследовательности и поиск наибольшей возрастающей подпоследовательности.

Вперёд

Решите оставшиеся задачи по этой теме в Яндекс.Контесте: https://contest.yandex.ru/contest/25896/problems/.

Когда закончите, переходите к финальному заданию ниже.

Финальное задание спринта 7

Вам предстоит решить две задачи и пройти код-ревью по ним.

Сдавайте решения в Яндекс.Контесте, а когда получите OK по обеим задачам, — запакуйте исходные файлы в один архив и загрузите на ревью. В начале каждого решения в комментарии:

- напишите объяснение так же, как в предыдущем спринте;
- укажите ID успешной посылки, чтобы ревьюер мог удостовериться, что решение рабочее.

Важно: выберите один язык программирования и сдавайте финальные задачи на ревью только на нём. Задачи, которые не требуют ревью, вы можете сдавать на любом из доступных языков.

Успехов!

Вперед