Задача «3-правильная очередь»

Краткое условие без легенды

Есть три вида событий — `A`, `B` и `C`. Один запрос может содержать только одну пару `AB`, `AC` и `BC` **(именно в таком порядке)**.

Запись происходит асинхронно, события запроса в очереди **могут лежать не последовательно**, то есть после записи первого события первого запроса, могло записаться событие от другого запроса.

Требуется понять, возможно ли корректно разделить события на пары.

Пример

Для набора событий «BAABCC» один из возможных вариантов — набор пар с индексами:

1. (1, 5) — «BC».
2. (2, 4) — «AB».
3. (3, 6) — «AC».

Разбор

Давайте сначала придумаем решение и объясним его неформально, а потом строго докажем его корректность.

Первым делом посчитаем количество всех трех переменных — $count_A, count_B, count_C$. С помощью этих переменных мы можем посчитать, какое количество пар AB, AC, BC должно быть в ответе.

Для этого нужно решить следующую систему уравнений:

1. $count_{AB} + count_{AC} = count_A$.
2. $count_{AB} + count_{BC} = count_B$.

3. $count_{AC} + count_{BC} = count_C$.

Нас интересует только $count_{BC}$, поэтому решением будет $count_{BC} = count_B - (n - count_C * 2)/2$.

Заведем два дека — один для позиций A($positions_a$), другой для позиций B($positions_b$). Будем идти с начала строки до ее конца, на каждой позиции у нас есть три варианта, что делать:

- Если мы встретили A, то просто положим ее в $positions_a$.
- Если мы встретили B, то просто положим ее в $positions_b$.
- Если же мы встретили C, то нужно понять, к кому ее поставить в пару.

  Если $positions_b$ не пуст и при этом $count_{bc}$ меньше количества уже набранных пар BC, то возьмем B с наименьшей позицией из него и поставим в пару к этой C. На самом деле, нам всегда выгоднее брать C в пару к B, чем в пару к A. Мы чуть позже это докажем.

  Если $positions_a$ не пуст, то возьмем A с наибольшей позицией из него и поставим в пару к этой C

  Если оба пусты, то значит, что к этой C никого в пару поставить нельзя, и следовательно ответ — «нельзя».

Если после этого цикла из оставшихся A и B можно собрать пары, то ответ — «можно», а иначе — «нельзя».

Для того, чтобы это проверить мы должны:

1. Проверить, что размеры $positions_a$ и $positions_b$ равны.
2. Проверить, что $positions_a[i] < positions_b[i]$ для всех i.

Пример решения

Пусть есть достаточно большая строка: $BBAABABCCABACCCBCCBB$.

Давайте на ее примере разберем, как этот алгоритм будет работать.

Первым делом мы посчитаем $count_A = 5$, $count_B = 8$, $count_C = 7$.

По формуле посчитаем $count_{BC} = count_B - (n - count_C * 2)/2 = 8 - (20 - 7 * 2)/2 = 8 - 3 = 5$.

- $positions_a = \{\}$, $positions_b = \{\}$, $count_{BC} = 5$, встречаем символ B, кладем его индекс в $positions_b$.
- $positions_a = \{\}$, $positions_b = \{0\}$, $count_{BC} = 5$, встречаем символ B, кладем его индекс в $positions_b$.
- $positions_a = \{\}$, $positions_b = \{0, 1\}$, $count_{BC} = 5$, встречаем символ A, кладем его индекс в $positions_a$.
- $positions_a = \{2\}$, $positions_b = \{0, 1\}$, $count_{BC} = 5$, встречаем символ A, кладем его индекс в $positions_a$.
- $positions_a = \{2, 3\}$, $positions_b = \{0, 1\}$, $count_{BC} = 5$, встречаем символ B, кладем его индекс в $positions_b$.
- $positions_a = \{2, 3\}$, $positions_b = \{0, 1, 4\}$, $count_{BC} = 5$, встречаем символ A, кладем его индекс в $positions_a$.
- $positions_a = \{2, 3, 5\}$, $positions_b = \{0, 1, 4\}$, $count_{BC} = 5$, встречаем символ B, кладем его индекс в $positions_b$.
- $positions_a = \{2, 3, 5\}$, $positions_b = \{0, 1, 4, 6\}$, $count_{BC} = 5$, встречаем символ C, так как $positions_b$ не пуста и $cnt_{bc} > 0$, то достаем B с минимальным индексом и формируем пару с ним. Пара — $\{0, 7\}$.
- $positions_a = \{2, 3, 5\}$, $positions_b = \{1, 4, 6\}$, $count_{BC} = 4$, встречаем символ C, так как $positions_b$ не пуста и $count_{BC} > 0$,

то достаем B с минимальным индексом и формируем пару с ним. Пара — $\{1, 8\}$.

- $positions_a = \{2, 3, 5\}$, $positions_b = \{4, 6\}$, $count_{BC} = 3$, встречаем символ A, кладем его индекс в $positions_a$.
- $positions_a = \{2, 3, 5, 9\}$, $positions_b = \{4, 6\}$, $count_{BC} = 3$, встречаем символ B, кладем его индекс в $positions_b$.
- $positions_a = \{2, 3, 5, 9\}$, $positions_b = \{4, 6, 10\}$, $count_{BC} = 3$, встречаем символ A, кладем его индекс в $positions_a$.
- $positions_a = \{2, 3, 5, 9, 11\}$, $positions_b = \{4, 6, 10\}$, $count_{BC} = 3$, встречаем символ C, так как $positions_b$ не пуста и $count_{BC} > 0$, то достаем B с минимальным индексом и формируем пару с ним. Пара — $\{12, 4\}$.
- $positions_a = \{2, 3, 5, 9, 11\}$, $positions_b = \{6, 10\}$, $count_{BC} = 2$, встречаем символ C, так как $positions_b$ не пуста и $count_{BC} > 0$, то достаем B с минимальным индексом и формируем пару с ним. Пара — $\{13, 6\}$.
- $positions_a = \{2, 3, 5, 9, 11\}$, $positions_b = \{10\}$, $count_{BC} = 1$, встречаем символ C, так как $positions_b$ не пуста и $count_{BC} > 0$, то достаем B с минимальным индексом и формируем пару с ним. Пара — $\{14, 10\}$.
- $positions_a = \{2, 3, 5, 9, 11\}$, $positions_b = \{\}$, $count_{BC} = 0$, встречаем символ B, кладем его индекс в $positions_b$.
- $positions_a = \{2, 3, 5, 9, 11\}$, $positions_b = \{15\}$, $count_{BC} = 0$, встречаем символ C, так как $positions_b$ не пуста, но $count_{BC} = 0$, то достаем A с максимальным индексом и формируем пару с ним. Пара — $\{16, 11\}$.
- $positions_a = \{2, 3, 5, 9\}$, $positions_b = \{15\}$, $count_{BC} = 0$, встречаем символ C, так как $positions_b$ не пуста, но $count_{BC} =$

0, то достаем A с максимальным индексом и формируем пару с ним. Пара — $\{17, 9\}$.

- $positions_a = \{2, 3, 5\}$, $positions_b = \{15\}$, $count_{BC} = 0$, встречаем символ B, кладем его индекс в $positions_b$.
- $positions_a = \{2, 3, 5\}$, $positions_b = \{15, 18\}$, $count_{BC} = 0$, встречаем символ B, кладем его индекс в $positions_b$.

Получили $positions_a = \{2, 3, 5\}$, $positions_b = \{15, 18, 19\}$, их размеры равны и при этом $\forall_i : positions_a[i] < positions_b[i]$.

Следовательно ответ — «можно».

Корректность

Одним из главных моментов нашего решения является утверждение про порядок выбора, а именно то, что мы всегда стараемся выбрать в первую очередь B с минимальным индексом. Если же свободных B нет, то стараемся выбрать A с максимальным индексом.

Докажем по пунктам:

- Докажем сначала, почему выгоднее выбрать минимальное B. Допустим, что наше решение выбрало пару $BC = \{x_1, y_1\}$, а какое-то другое решение выбрало $BC = \{x_2, y_1\}$, при этом $x_1 < x_2$. Тогда в этом решении для нашего B была выбрана либо пара $AB = \{x_3, x_1\}$, либо пара $BC = \{x_1, y_2\}$. Так как $x_3 < x_1 < x_2$, то мы можем без проблем поменять обе B местами(x_2 и x_1), то же самое и в случае смены местами A и B.
- Затем докажем, почему выгоднее выбрать максимальное A. Допустим, что наше решение выбрало пару $AC = \{x_1, y_1\}$, а какое-то другое решение выбрало $AC = \{x_2, y_1\}$(при этом $x_2 < x_1$. Тогда в этом решении для нашего A была выбрана либо пара $AB = \{x_1, x_3\}$, либо пара $AC = \{x_1, y_2\}$. Так как $x_2 < x_1 < x_3$, то мы можем без проблем поменять A местами. То есть логика такая же, как и в предыдущем кейсе.

- Затем докажем, что выгоднее выбирать B, а не A.

  Если доказывать неформально, то все просто — в конце мы проверяем два массива по позициям, а следовательно нам выгодно, чтобы в получившихся массивах все A были как можно меньше, а B как можно больше.

  Так как на каждом шаге мы выбираем минимальные B и максимальные A, то выбирать B нам выгоднее, чем A.

  Более формально — допустим, в нашем решении мы выбрали $BC = \{x_1, y_1\}$, а в каком-то другом $AC = \{x_2, y_1\}$, тогда B может быть либо в паре с каким-то другим C и тогда мы без проблем можем поменять их пары (поскольку в нашем решении мы всегда выбираем минимальное), либо в паре с какой-то другой A. Во втором случае, мы продолжаем менять связь в нашем решении (то есть из пары $ABBC$ делаем $ABBC$), до тех пор пока не дойдем до B, связанной с C (так как количество пар BC всегда одно и тоже) и следовательно опять поменяем их местами.

Время работы и память

В начале нам нужно посчитать количество каждого символа в строке, это будет работать за линейное время.

Затем по формуле мы считаем количество пар, это O(1).

Затем мы проходимся по всей строке, поддерживая при этом дек символов B, A, для каждой позиции мы либо вставляем в дек, либо удаляем, а также делаем константное количество сравнений, следовательно цикл также работает за O(n).

Финальная часть(сравнение результирующих деков) также работает за O(n), следовательно суммарная асимптотика - O(n).

Дополнительная память также линейная, так как мы поддерживаем деки элементов.

Задача «Три банка, три валюты»

Краткое условие без легенды

Банки A, B и C предлагают обмен валюты.

Каждый банк меняет рубли, доллары и евро по своему курсу. Вы можете обращаться в каждый из банков не более одного раза. Под одним обращением понимается 1 операция обмена валюты. Нельзя внутри одного банка совершить несколько операций обмена валют.

Вам необходимо перевести 1 рубль в максимально возможное количество долларов.

Пример теста

В данно задаче важно было аккуратно разобрать входные данные, давайте полностью рассмотрим тест из условия:

- 100:1 — 1 банк (курс обмена рублей на доллары).
- 100:1 — 1 банк (курс обмена рублей на евро).
- 1:100 — 1 банк (курс обмена долларов на рубли).
- 3:2 — 1 банк (курс обмена долларов на евро).
- 1:100 — 1 банк (курс обмена евро на рубли).
- 2:3 — 1 банк (курс обмена евро на доллары).
- 100:1 — 2 банк (курс обмена рублей на доллары).
- 100:1 — 2 банк (курс обмена рублей на евро).
- 1:100 — 2 банк (курс обмена долларов на рубли).
- 3:2 — 2 банк (курс обмена долларов на евро).
- 1:100 — 2 банк (курс обмена евро на рубли).

- 2:3 — 2 банк (курс обмена евро на доллары).
- 100:1 — 3 банк (курс обмена рублей на доллары).
- 100:1 — 3 банк (курс обмена рублей на евро).
- 1:100 — 3 банк (курс обмена долларов на рубли).
- 3:2 — 3 банк (курс обмена долларов на евро).
- 1:100 — 3 банк (курс обмена евро на рубли).
- 2:3 — 3 банк (курс обмена евро на доллары).

Итого получаем:

- 1 банк:

currency	rur	usd	eur
rur	1:1	100:1	100:1
usd	1:100	1:1	3:2
eur	1:100	2:3	1:1

- 2 банк:

currency	rur	usd	eur
rur	1:1	100:1	100:1
usd	1:100	1:1	3:2
eur	1:100	2:3	1:1

- 3 банк:

currency	rur	usd	eur
rur	1:1	100:1	100:1
usd	1:100	1:1	3:2
eur	1:100	2:3	1:1

Можно заметить, что в данном тесте у всех банков курсы одинаковые, а следовательно, на финальное количество долларов влияет только выбор и порядок пар валют для обмена.

Проверив все возможные порядки для этого теста, можно увидеть, что здесь больше всего долларов мы получим после следующего порядка: $RUR \Rightarrow EUR \Rightarrow USD$.

Разбор

Ограничения в этой задаче были достаточно маленькими, поэтому можно было решать задачу любым достаточно комфортным для вас способом.

Одним из возможных решений будет рекурсивная функция $find\_biggest\_amount(amount, currency, used\_banks)$, где $amount$ — текущее количество валюты $currency$, а $used\_banks$ — номера уже использованных банков.

На каждом шаге мы будем перебирать следующий банк и для каждого банка выбирать один из трех возможных шагов.

Например, если у нас были рубли, то шаги могут быть следующими:

1. Поменять в рубли (в данном случае это тоже самое, что и не сделать ничего).

2. Поменять в доллары.

3. Поменять в евро.

Пример решения

Покажем, как будет работать наше решение на тесте из условия.

1. Изначально мы начинаем с 1 рублем, то есть вызываем функцию $find\_biggest\_amount(1, rur, \{\})$.

Начинаем перебор банков с первого банка.

Начинаем перебор возможных операций с перевода в рубли.

Вызывем функцию $find\_biggest\_amount(1, rur, \{1\})$.

2. Обрабатываем вызов $find\_biggest\_amount(1, rur, \{1\})$.

Начинаем перебор банков с второго банка.

Начинаем перебор возможных операций с перевода в рубли.

Вызывем функцию $find\_biggest\_amount(1, rur, \{1, 2\})$.

3. Обрабатываем вызов $find\_biggest\_amount(1, rur, \{1, 2\})$.

Начинаем перебор банков с третьего банка.

Начинаем перебор возможных операций с перевода в рубли.

Вызывем функцию $find\_biggest\_amount(1, rur, \{1, 2, 3\})$.

4. Обрабатываем вызов $find\_biggest\_amount(1, rur, \{1, 2, 3\})$. Так как больше свободных банков нет, то это конечный вызов и возвращаемся в прошлый шаг перебора.

5. Вернулись к вызову $find\_biggest\_amount(1, rur, \{1, 2\})$.

Продолжаем перебор банков с третьего банка.

Продолжаем перебор возможных операций с перевода в доллары.

Вызывем функцию $find\_biggest\_amount(0.01, usd, \{1, 2, 3\})$.

6. Обрабатываем вызов $find\_biggest\_amount(0.01, usd, \{1, 2, 3\})$. Так как валюта — доллар, обновляем ответ. Так как больше свободных банков нет, то это конечный вызов и возвращаемся в прошлый шаг перебора.

7. Вернулись к вызову $find\_biggest\_amount(1, rur, \{1, 2\})$.

Продолжаем перебор банков с третьего банка.

Продолжаем перебор возможных операций с перевода в евро.

Вызывем функцию $find\_biggest\_amount(0.01, eur, \{1, 2, 3\})$.

8. Обрабатываем вызов $find\_biggest\_amount(0.01, eur, \{1, 2, 3\})$. Так как больше свободных банков — нет, то это конечный вызов и возвращаемся в прошлый шаг перебора.

9. Вернулись к вызову $find\_biggest\_amount(1, rur, \{1, 2\})$.

Мы все перебрали, поэтому возвращаемся в $find\_biggest\_amount(1, rur, \{1\})$.

Давайте также сразу и посмотрим путь, дающий лучший ответ:

1. Изначально мы начинаем с 1 рублем, то есть начинаем с вызова $find\_biggest\_amount(1, rur, \{\})$.

Начинаем перебор банков с первого банка.

Дойдем до операции перевода в евро.

Вызывем функцию $find\_biggest\_amount(0.01, eur, \{1\})$.

2. Обрабатываем вызов $find\_biggest\_amount(0.01, eur, \{1\})$.

Начинаем перебор банков с второго банка.

Дойдем до операции перевода в доллар.

Вызывем функцию $find\_biggest\_amount(0.015, usd, \{1, 2\})$.

3. Обрабатываем вызов $find\_biggest\_amount(0.015, usd, \{1, 2\})$.

Начинаем перебор банков с третьего банка.

Начинаем перебор возможных операций с перевода в рубли.

Вызывем функцию $find\_biggest\_amount(0.015, usd, \{1, 2, 3\})$.

4. Обрабатываем вызов $find\_biggest\_amount(0.015, usd, \{1, 2, 3\})$. Это и будет ответ.

Корректность решения

Мы перебираем все возможные варианты, следовательно, мы получим все возможные способы перевода валют, а следовательно, и наилучший ответ.

Важное про вещественные числа

Вообще в данной задаче прекрасно работали и вещественные числа, так как требовалась достаточно небольшая точность, поэтому решения с float или double должны были работать.

Также можно было использовать decimal или самому его написать, если хотелось большей точности.

Время работы и память

Наше решение будет перебирать все банки и все валюты, из чего мы получим $O(banks! \cdot currencies^{banks})$, что в нашем случае будет $O(1)$, дополнительной памяти на рекурсию требуется столько же.

Задача «3-Покер»

Краткое условие без легенды

У нас есть карты от 2 до туза и все 4 масти.

Игра происходит следующим образом:

1. Изначально все n игроков получают по две карты из колоды.
2. После этого на стол выкладывается одна карта из той же колоды.
3. Выигрывают те игроки, у которых собралась самая старшая комбинация.

Комбинации от большей к меньшей:

- если две карты у игрока в руке и карта на столе имеют одинаковое значение, игрок собрал комбинацию '*Сет со значением x*'.
- если из двух карт у игрока в руке и карты на столе можно выбрать две карты с одинаковым значением x, игрок собрал комбинацию '*Пара со значением x*'.
- иначе, берется карта с самым старшим значением из двух карт у игрока в руке и карты на столе, тогда игрок собрал комбинацию '*Старшая карта x*'.

Если одинаковая самая старшая комбинация есть у нескольких игроков, все они объявляются выигравшими.

Вам даны все карты n игроков. Требуется определить при каких картах на столе выигрывает первый игрок.

Пример теста

- У первого человека на руках карты TS и TC, то есть 10 пики (ten spades) и 10 треф (ten clubs).
- У второго человека на руках карты AD и AH, то есть туз буби (ace diamonds) и туз червей (ace hearts).

Для победы подходит карта TD, так как тогда у первого игрока будет '*Сет со значением T*', а у второго '*Пара со значением A*' и первый игрок выигрывает.

Для победы не подходит карта JD, так как тогда у первого игрока будет '*Пара со значением J*', а у второго '*Пара со значением A*' и первый игрок проигрывает.

Разбор

Самым простым решением будет проверить для каждой карты, кто выиграет, если сейчас эта карта окажется на столе.

Тогда решение разбивается на две части:

1. Как правильно перебрать все карты?
2. Как проверить, кто выиграет при какой-то карте?

Как правильно перебрать все карты?

Достаточно простым в реализации является следующий вариант:

Заведем массив всех значений (value):

2, **3**, **4**, **5**, **6**, **7**, **8**, **9**, **T**, **J**, **Q**, **K**, **A**.

Заведем массив всех мастей (suit):

S, **C**, **D**, **H**.

Также заведем словарь уже использованных карт, то есть карт, которые на руках у любого из игроков.

Будем перебирать карту как элемент из значений и элемент из мастей, затем нам нужно проверить, что эта карта еще не использована, при помощи словаря уже использованных карт.

Пример перебора

В тесте из условия словарь использованных карт:
$\{TS, TC, AD, AH\}$.
Промоделируем перебор:

1. value = 2, suit = S, card = 2S, карта еще не использована.
2. value = 3, suit = S, card = 3S, карта еще не использована.

   ...

10. value = T, suit = S, card = TS, карта использована, не рассматриваем.
11. value = J, suit = S, card = JS, карта еще не использована.
12. value = Q, suit = S, card = QS, карта еще не использована.
13. value = K, suit = S, card = KS, карта еще не использована.
14. value = A, suit = S, card = AS, карта еще не использована.
15. value = 2, suit = C, card = 2C, карта еще не использована.

    ...

25. value = T, suit = C, card = TC, карта использована, не рассматриваем.

    ...

51. value = K, suit = H, card = KH, карта еще не использована.
52. value = A, suit = H, card = AH, карта использована, не рассматриваем.

Таким образом мы получили список карт для проверки.

Как проверить, кто выиграет при какой-то карте?

Для карт можно завести отдельный класс или работать с ними, как со строками. Пример класса на Go:

```
type Value int

const (
  Value2 Value = iota
  Value3
  Value4
  Value5
  Value6
  Value7
  Value8
  Value9
  ValueT
  ValueJ
  ValueQ
  ValueK
  ValueA
)

func (v Value) String() string {
  return [...]string{"2", "3", "4", "5", "6", "7", "8", "9", "T", "J
    ", "Q", "K", "A"}[v]
}

type Cars struct {
  Value    Value
    Suit     char
}
```

Сделаем функцию $check\_combination(card_1, card_2, card_3)$, которая по трем картам возвращает комбинацию.

Первым делом заметим, что в комбинациях никак не участвует масть, поэтому нам достаточно проверять только значение.

Также очень удобно сделать отдельную структуру для комбинации: Combination, которую будет содержать количество одинаковых карт и значение, вот пример на языке Go:

```
type Combination struct {
  Value    Value
    Count    int
}

func (c Combination) Less(other Combination) bool {
```

```
  if c.Count != other.Count {
    return c.Count < other.Count
  }
  return c.Value < other.Value
}
```

Проверка комбинации же выглядит следующим образом:

1. Если $card_1.Value = card_2.Value = card_3.Value$, то возвращаем $Combination(card_1.Value, 3)$.
2. Если $card_1.Value = card_2.Value$ или $card_1.Value = card_3.Value$, то возвращаем $Combination(card_1.Value, 2)$.
3. Если $card_2.Value = card_3.Value$, то возвращаем $Combination(card_2.Value, 2)$.
4. Возвращаем $Combination(max(card_1.Value, card_2.Value, card_3.Value), 1)$.

Пример проверки

Для тройки TS, TC, TD мы пройдем следующий путь:

1. $card_1.Value = card_2.Value = card_3.Value = T$, следовательно возвращаем $Combination(T, 3)$.

Для тройки AD, AH, TD мы пройдем следующий путь:

1. $card_1.Value = card_2.Value \neq card_3.Value$.
2. $card_1.Value = card_2.Value = A$, возвращаем $Combination(A, 2)$.

$Combination(A, 2) < Combination(T, 3)$, следовательно первый игрок выиграл.

Время работы и память

Максимально возможное количество карт — 52.

Проверка, кто выиграл при выкладывании карты, работает за O(1).

Таким образом все решение работает за O(1).

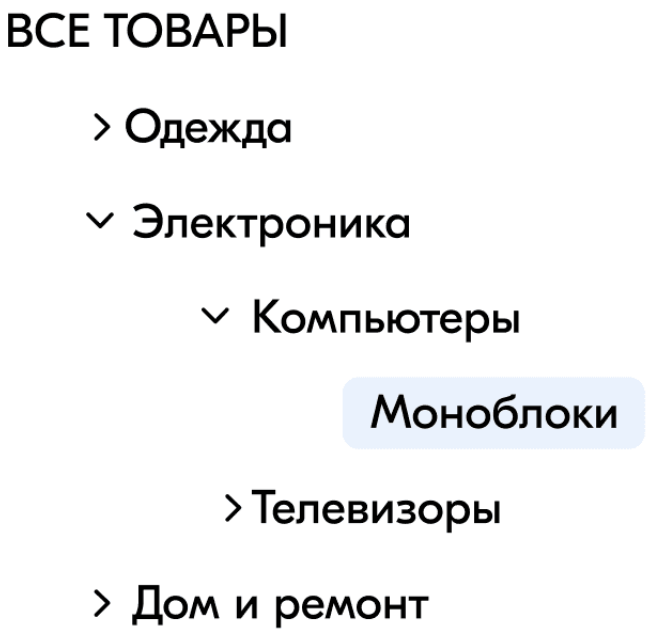
Дополнительная память нам требуется только на хранение трех карт и получившейся комбинации, следовательно, дополнительной памяти также будет O(1).

Задача «JSON категории»

Краткое условие без легенды

У каждого товара есть категория. При этом, у некоторых категорий есть дочерние категории.

Назовем деревом категорий такую сущность:



Ваша задача — построить дерево категорий.

Дана информация об отношениях родительских и дочерних категорий в виде JSON-массива. Каждый элемент массива является словарем, с полями `name` (название категории), `id` (числовой идентификатор категории) и `parent` (числовой идентификатор родительской категории).

Известно, что корневая категория имеет нулевой идентификатор и не имеет идентификатора родительской категории.

По данной информации постройте дерево категорий в виде JSON-словаря. Словарь для каждой категории должен иметь поля `name`,

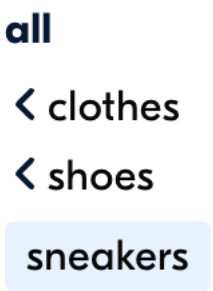
`id` и массив `next`, состоящий из таких же словарей для дочерних категорий.

Пример теста

В тесте из условия был такой JSON:

```
[
   {
     "id":0,
     "name":"all"
   },
   {
      "id":1,
      "name":"clothes",
      "parent":0
   },
   {
      "id":2,
      "name":"shoes",
      "parent":0
   },
   {
      "id":55,
      "name":"sneakers",
      "parent":2
   }
]
```

Этому JSON соотвествует следующее дерево категоий:



Финальное же представление дерева в виде JSON будет таким:

```
[
    {
        "id": 0,
        "name": "all",
        "next": [
            {
                "id": 1,
                "name": "clothes",
                "next":[]
            },
            {
                "id": 2,
                "name": "shoes",
                "next": [
                    {
                        "id": 55,
                        "name": "sneakers",
                        "next":[]
                    }
                ]
            }
        ]
    }
]
```

Разбор

В этой задаче можно было написать решение и с самостоятельным составлением JSON, но можно было использовать готовые библиотеки, разрешенные в задаче ("encoding/json"в Go или ее аналоги в других языках).

Так что самым важным пунктом для нашего решения будет вопрос о том как аккуратно все это реализовать.

Первым делом было удобно реализовать специальные структуры для категории и для целого дерева категорий.

```
type Category struct {
  ID       int    `json:"id"`
  Name     string `json:"name"`
  ParentID int    `json:"parent"`
}

type Tree struct {
  ID   int     `json:"id"`
```

```
  Name string  `json:"name"`
  Next []*Tree `json:"next"`
}
```

Нам нужно получить из входных данных что-то достаточно осмысленное. Для этого мы построчно считаем входной файл, сложим результат в одну строку и из JSON получим массив категорий. Пример кода на Go:

```
inputString := ""
for i := 0; i < n; i++ {
    line, _ := in.ReadString('\n')
    inputString += line
}

categories := make([]Category, 0)
json.Unmarshal([]byte(inputString), &categories)
```

Таким образом мы получим массив категорий, с которым позже сможем комфортно работать.

Создадим словарь вершин дерева, то есть $map[int] * Tree$, который по id Tree будет возвращать соответсвующую структуру.

Нам нужно собрать из массива категорий дерево, это можно сделать с помощью dfs или с помощью цикла.

В цикле мы будем класть в tree.next все вершины, у которых parent = tree.id.

```
for _, i := range arr {
    t := new(Tree)
    t.ID = i.ID
    t.Name = i.Name
    t.Next = make([]*Tree, 0)
    treeByNode[i.ID] = t
}
for _, i := range arr {
    if i.ID != 0 {
        treeByNode[i.ParentID].Next = append(treeByNode[i.ParentID].
   Next, treeByNode[i.ID])
    }
}
```

Положим полученное дерево в json и выведем ответ:

```
bytes, _ := json.Marshal(treeByNode[0]) // 0 - root category
```

Пример решения

Разберем работу решения на тесте из условия.

После парсинга входных данных мы получим массив категорий:

```
categories = [
    Category{ID: 0, Name: "all", ParentID: null},
    Category{ID: 1, Name: "clothes", ParentID: 0},
    Category{ID: 2, Name: "shoes", ParentID: 0},
    Category{ID: 55, Name: "sneakers", ParentID: 2},
]
```

После создания дерева мы получим следующее:

```
treeByNode = {
    0: *Tree{ID: 0, Name: "all", next: []},
    1: *Tree{ID: 1, Name: "clothes", next: []},
    2: *Tree{ID: 2, Name: "shoes", next: []},
    55: *Tree{ID: 55, Name: "sneakers", next: []},
}
```

Финальный же цикл сделает следующее:

1. ```
   i.id = 0; continue;
   ```

2. ```
   i.id = 1; i.ParentID = 0;
   tree[treeByNode[0]].next.append(treeByNode[1]);
   tree[treeByNode[0]].next = [treeByNode[1]];
   ```

3. ```
   i.id = 2; i.ParentID = 0;
   tree[treeByNode[0]].next.append(treeByNode[2]);
   tree[treeByNode[0]].next = [treeByNode[1], treeByNode[2]];
   ```

4. ```
   i.id = 55; i.ParentID = 2;
   tree[treeByNode[2]].next.append(treeByNode[55]);
   tree[treeByNode[2]].next = [treeByNode[55]];
   ```

Мы получим такое дерево:

```
treeByNode = {
    0: *Tree{ID: 0, Name: "all", next: [treeByNode[1], treeByNode
    [2]]},
    1: *Tree{ID: 1, Name: "clothes", next: []},
    2: *Tree{ID: 2, Name: "shoes", next: [treeByNode[55]]},
    55: *Tree{ID: 55, Name: "sneakers", next: []},
}
```

Время работы и память

Encode и decode JSON работают за линейное от его размера время.

Дерево мы строим также за линейное время, следовательно итоговая асимптотика — $O(len(json))$.

Дополнительной памяти нам также требуется — $O(len(json))$, так как нам требуется сохранить финальный JSON, а также требуется строка с исходным JSON.