Дискретный анализ, 2 семестр. Ответы на вопросы из лекций (без доказательств). В данном конспекте встречаются ошибки, также он не полный!

Оглавление

Блок I. Динамическое программирование и жадные алгоритмы

1. Динамическое программирование. Основные этапы построения решения, оптимальная подструктура, перекрытие вспомогательных подзадач, мемоизация.

Динамическое программирование

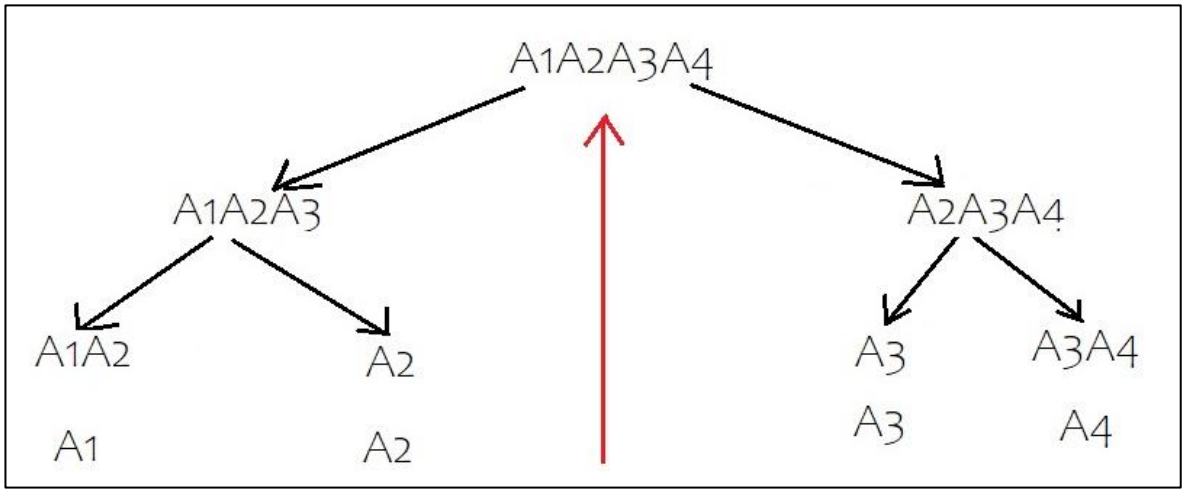
1. … - это **подход** в решении задач, в котором мы пытаемся решить задачу, **разбив её на подзадачи**, причём **такие** же, но **меньшего** размера и в отличие от подхода «разделяй и властвуй» подзадачи **могут пересекаться**.
   Этот метод предусматривает **использование накопленных данных**, поэтому компоновка этих зависимых задач должна занимать меньше времени.
2. … решает задачи **оптимизации** (минимизации, максимизации).
3. … не всегда позволяет решить задачи всеми способами, а только одним (каким – очень зависит от реализации).
   ДМ – это **НЕ один метод**, это подход, **парадигма** (есть большая задачи, дробим на маленькие. А как? И так, и сяк, надо самим подбирать стратегию)

Основные этапы построения решения:

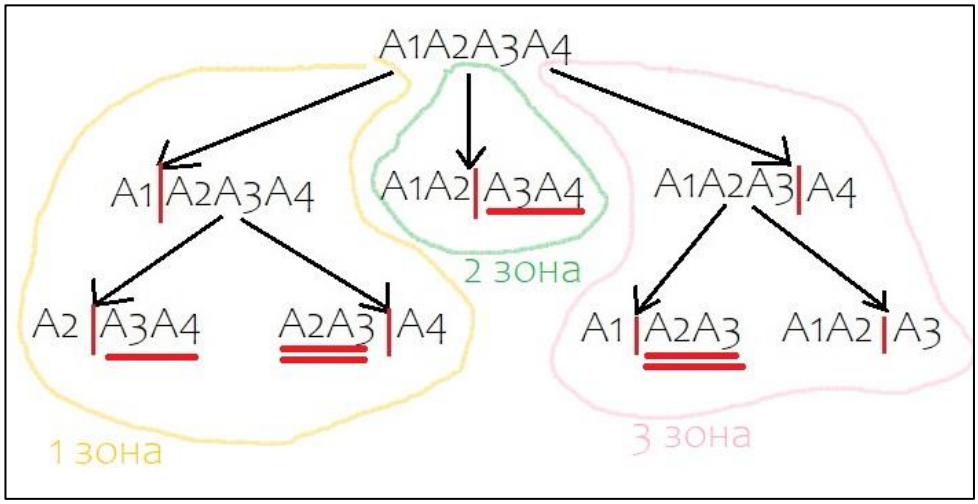
1. Эта задача на **оптимизацию** (вам дано что-то, и в этом чём-то надо найти как это что-то решается оптимальным образом – обычно минимум или максимум).
2. Если можно **большое решение** задач расписать **через решение меньших** таких же задач, можно их **объединить и получить решение** (т. е. кажется, что есть рекуррентная структура). Тогда есть метод нисходящего анализа.
3. Можно ли «вывернуть» эту задачу, то есть **из решения маленьких задач можно получить большое** (восходящий анализ).
4. Понять, что **общих ресурсов** у задач **не будет**.

Нисходящий и восходящий анализ:

1. **Нисходящий** анализ - решение задач можно расписать через решение меньших таких же задач, после чего можно их объединить и получить итоговое решение.
   = **Решаем от маленьких задач к большим.**
2. **Восходящий** анализ - мы знаем, как получит ответ на самых ранних этапах, а от них можем решить большие задачи.
   = **Разбить большую задачу на маленькие, решить маленькие и из них составить большую.**
3. Разница на примере задачи о перемножении четырёх матриц (вопрос 3).
   Нарисуем имитацию дерева решений.
   1) Восходящий анализ.
      Сначала считаем A1, A2, A3, A4, потом A1A2, A3A4, потом A1A2A3, потом A2A3A4:

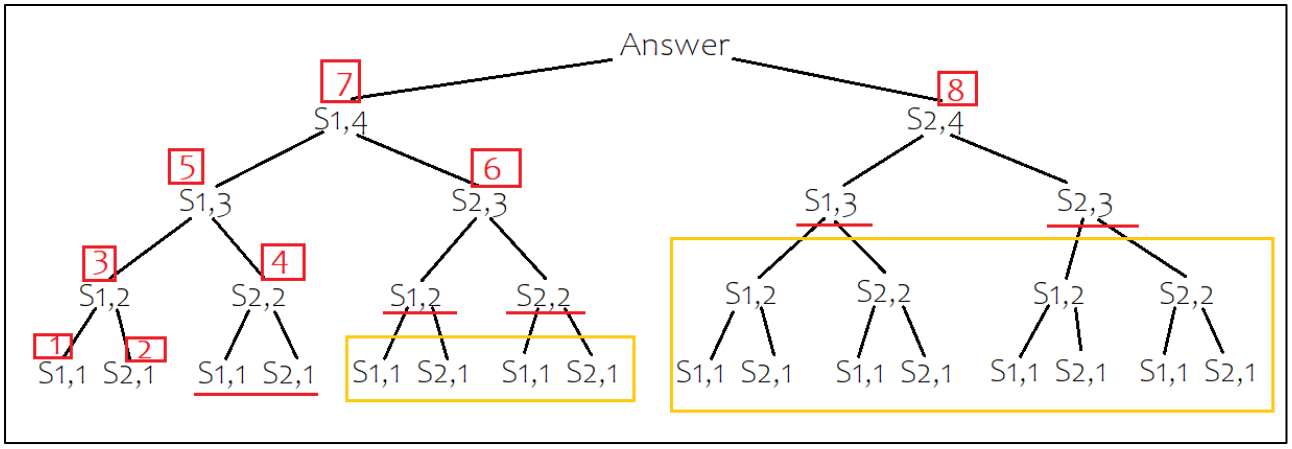


2) Нисходящий анализ: сначала считаем зону 1: A_3A_4, A_2A_3, $A_2A_3A_4$ - потом считаем зону 2, зону 3 и т. д:



Мемоизация:

1. … - **оптимизация** для анализа, заключающаяся в **запоминании промежуточных результатов** и, благодаря этому, **устранении повторных расчётов** одних и тех же значений.
2. … - **запоминание** того, что было на предыдущих шагах и **использование** этого как-то в решении. Процесс проверки, была ли решена задача, если да – переносим ответ, если нет – решаем и сохраняем его. То есть в чём идея: мы используем то, что уже решили в предыдущих зонах.
3. … на примере задачи с конвейерами (вопрос 2). Визуализируем решение для задачи:

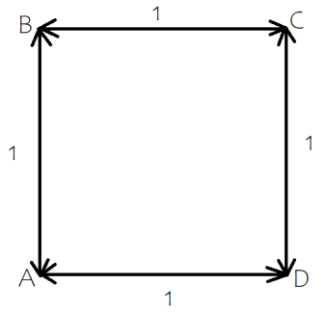


Благодаря мемоизации за счёт проверок мы можем не считать элементы, подчёркнутые красными и не смотреть даже их детей (в жёлтой рамке).

4. …, в которой хранятся **не все** результаты - *кэширование*.

Перекрытие вспомогательных задач:

1. Оптимальные подзадачи не должны шарить между собой ресурсы (а, допустим, только последние промежуточные решения).
2. Пример-объяснение:



Предположим, есть задача по нахождению кратчайшего пути из A в C.
У нас есть вариант пойти из A->B, из B->C. Получили две подзадачи.

- Путь из A->B можно записать так: A->D->C->B.
- Путь из B->C так: B->A->D->C.

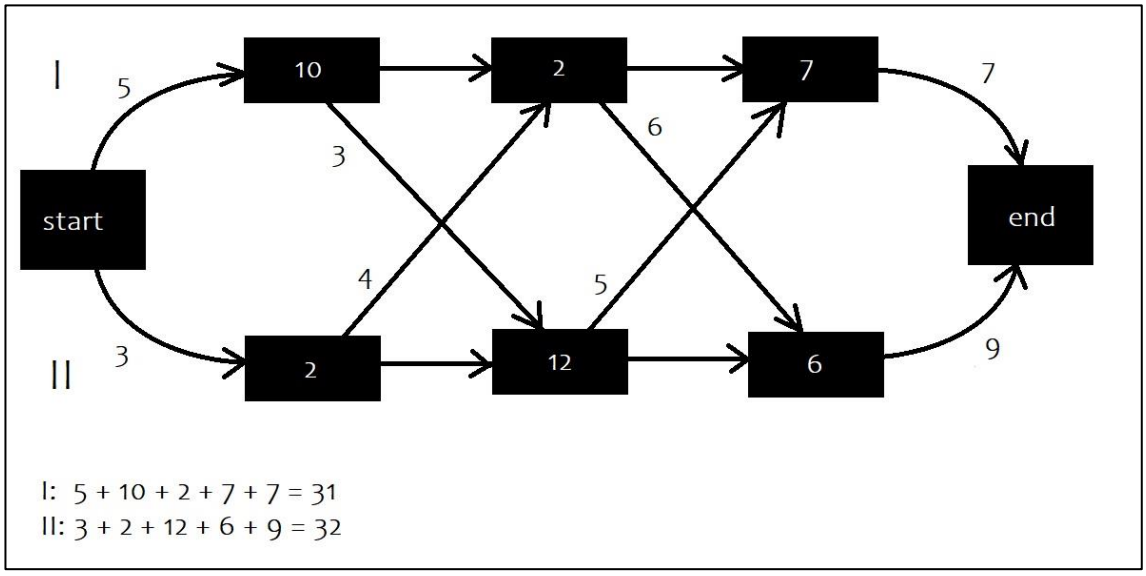
Получим A->C = A->D->C->B->A->D->C.
Мы получили, что мы дважды зашли в вершины A, C, D. Очевидно, что это неоптимальное решение.
Почему так получилось? Потому что наши подзадачи (A->B, B->C) использовали одни и те же ресурсы – вершины C и D. Поэтому эту задачу с помощью ДМ не решить.

2. Задача о расписании работы конвейера.

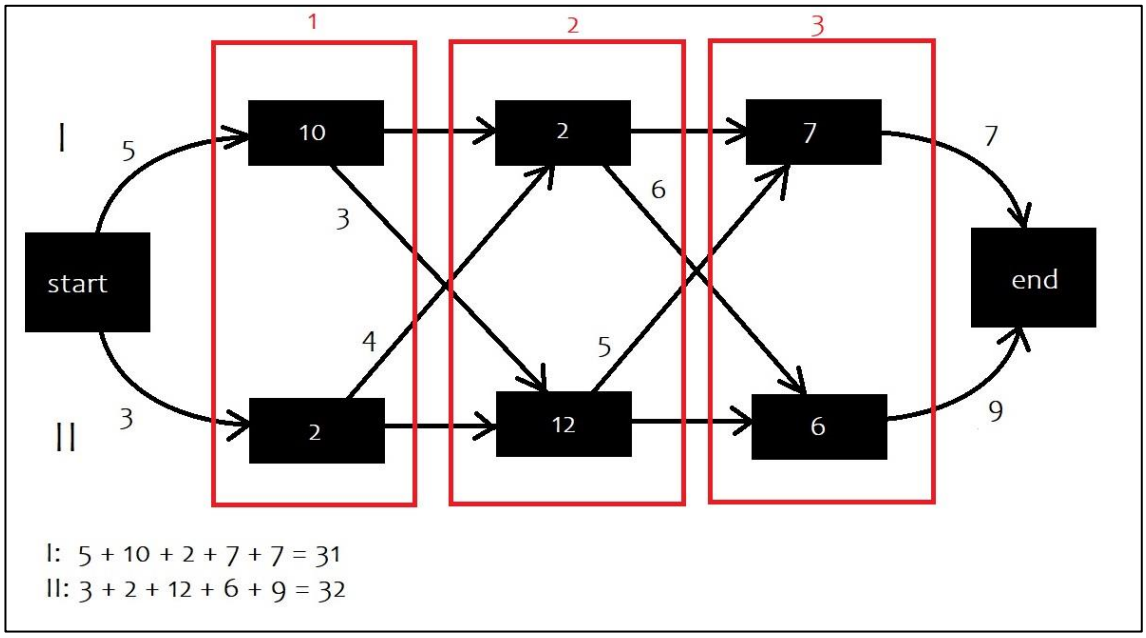
Ключ: равноправные этапы, два массива, подзадачи элементарны и зависимы, на каждой итерации выбираем минимальный возможный путь - оптимизация, min ($S_{I,i-1}$, $S_{II,i-1} + P_{i-1}$) + a_i, O(2n). В лоб – 2^n.

Дано:

Представим завод по производству лады калины. У нас есть два конвейера, на них одинаковое количество этапов. Этапы равноправны.



Пусть время, необходимое для транспортировки деталей пишем на рёбрах, а время, необходимое для этапов в узлах. Так как этапы равноправны, мы можем оправить детали на обработку во второй конвейер (например: 3(II) + 2(II) + 4(transport) + 2(I) + 7(I) + 7(I) = 25)



Найти:

Найти наименьшее затраченное время для обработки деталей (и путь).

Решение 1 – в лоб:

Решим задачу в лоб.

Пусть путь в I – 0, в II – 1.

Пример перебора: 5 + (10) + 3 + (12) +5 + (7) + (7). Будет 010.

Сложность:

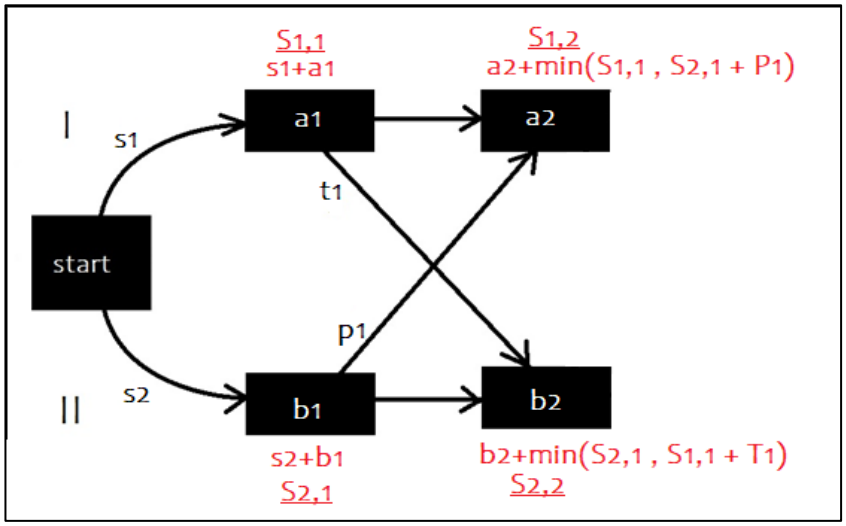
- У нас есть n (= 3) этапов. Всего у нас будет 2^n вариантов, как пройти по конвейеру и это необходимо сделать для каждого элемента, значит, сложность будет O (n *2^n).
- Если делать всё рекурсивно, то можно получить сложность O(2^n).

Решение 2 – с помощью ДП:

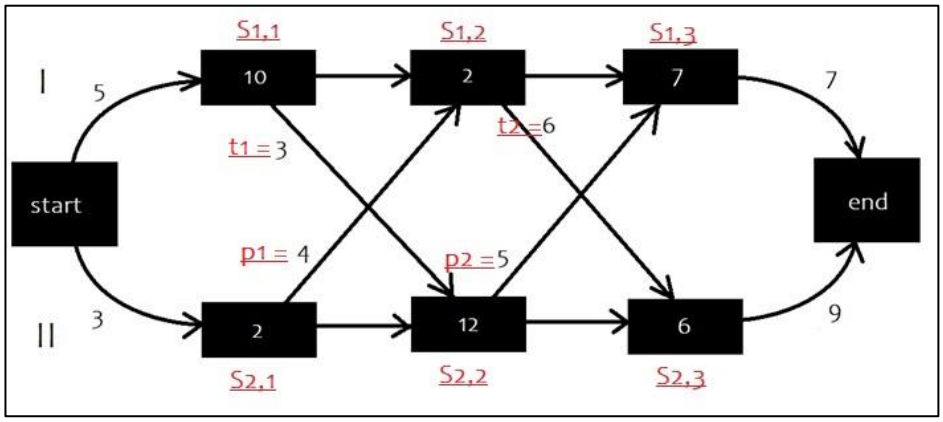
Можно решить эту же задачу методом динамического программирования.

Создадим массивы S_I, S_{II}.

$S_{I,i}$ - как пройти какой-то путь от первого элемента до i-го и остановиться на первом конвейере так, чтобы путь был оптимально возможным – это min ($S_{I,i-1}$ и $S_{II,i-1} + P_{i-1}$) + a_i



Задачи пересекаются, они не независимы! S_1 и S_2 решают одни и те же задачи.



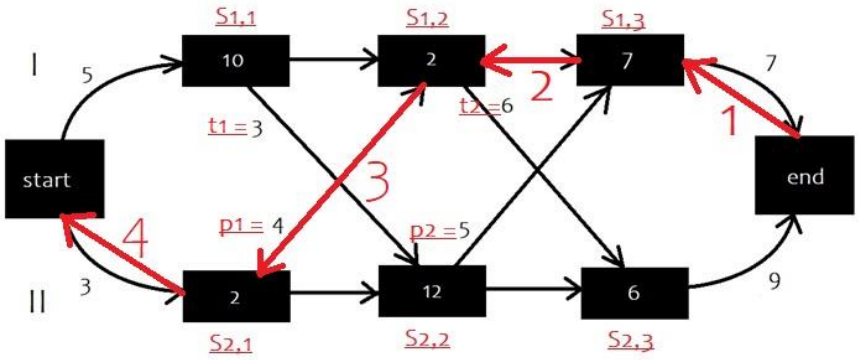
Посчитаем значения $S_{I,i}$ по формуле **$S_{I,i} = \min(S_{I,i-1}$ и $S_{II,i-1} + P_{i-1}) + a_i$**

$S_{I,k}$	k = 1	k = 2	k = 3	final
I = 1	1 действие $S_1 + a_1 = S_{1,1}$ 5 + 10 = 15	3 действие $\min(S_{1,1}, S_{2,1} + p_1) + a_2 = S_{1,2}$ min(15, 5 + 4) + 2 = 11	5 действие $\min(S_{1,2}, S_{2,2} + p_2) + a_3 = S_{1,3}$	7 действие $\min(S_{1,3} + e_1, S_{2,3} + e_2) = X$ min(18 + 7, 23 + 9) = 25

			min(11, 17 + 5) + 7 = 18	
I = 2	2 действие $S_2 + b_2 = S_{2,1}$ 3 + 2 = 5	4 действие $\min(S_{2,1}, S_{1,1} + t_1) + b_2 = S_{2,2}$ min(5, 15 + 3) + 12 = 17	6 действие $\min(S_{2,2}, S_{1,2} + t_2) + b_3 = S_{2,3}$ min(17, 11 + 6) + 6 = 23	

Как найти минимальный путь? Надо «пройтись» от результата назад и посмотреть минимумы и то, что мы выбирали – так мы поймём, как пришли туда.

$S_{I,k}$	k = 1	k = 2	k = 3	final
I = 1	1 действие $S_1 + a_1 = S_{1,1}$ 5 + 10 = 15	3 действие $\min(S_{1,1}, S_{2,1} + p_1) + a_2 = S_{1,2}$ min(15, 5 + 4) + 2 = 11	5 действие $\min(S_{1,2}, S_{2,2} + p_2) + a_3 = S_{1,3}$ min(11, 17 + 5) + 7 = 18	7 действие $\min(S_{1,3} + e_1, S_{2,3} + e_2) = X$ min(18 + 7, 23 + 9) = 25
I = 2	2 действие $S_2 + b_2 = S_{2,1}$ 3 + 2 = 5	4 действие $\min(S_{2,1}, S_{1,1} + t_1) + b_2 = S_{2,2}$ min(5, 15 + 3) + 12 = 17	6 действие $\min(S_{2,2}, S_{1,2} + t_2) + b_3 = S_{2,3}$ min(17, 11 + 6) + 6 = 23	



Итоговый путь отзеркалим: 3 + 2 + 4 + 2 + 7 + 7 = 25

Заметим! В теории мы могли бы решать всю эту задачу в обратном порядке (допустим, опишем результат – мы должны выбрать минимум из сумм пути и затрат на производстве. Для предыдущего шага – мы должны… и т. п.), однако это решение не даст линейную сложность, в отличие от анализа с начала. Поэтому ДМ – не один метод, это идея.

Идея динамического программирования тут:

> Мы взяли финальную точку – ответ, и мы знаем, что это минимум (экстремум) среди нескольких подзадач, каждая подзадача бьётся на такие задачи. Мы бьём задачи до тех пор, пока они не станут элементарными. Это метод – от большой к малым задачам – называется методом нисходящего анализа.
>
> В нашей задаче рекуррентное решение было – есть i-й этап какого-то конвейера и смотрим, как мы могли попасть в него.

Сложность решения этой задачи линейная, так как все точки мы считаем за линейное время (нам нужно посчитать 2*n по 2 расчёта на каждом из двух этапов за константу).

3. Задача о перемножении цепочки матриц.

Ключ: числа Каталана, сложность перемножения двух матриц - p * q * r, если конечный вариант оптимален, его подзадачи тоже оптимальны, считаем самые маленькие подзадачи (сначала * двух матриц, потом трёх и доходим до ответа), сложность $O(n^3)$ для 4-х матриц.

Дано:

Последовательность матриц $A_1 * A_2 … * A_n$ и их размеры.

Найти:

Расставить скобки между матрицами так, чтобы их перемножение было оптимальным (определить приоритет умножений).

Решение:

Пусть есть матрицы $A_1A_2A_3$, и их размеры – 10*5, 5*100, 100*50.

Сложность перемножения двух матриц $A_{p*q} * B_{q*r} = C_{p*r}$ будет p * q * r.

При перемножении трёх матриц так: $(A_{10*5} * A_{5*100}) * A_{100*50}$ - сложность будет 10 * 5 * 100 + 10 * 100 * 50 = 5000 + 50000 = 55000. А что, если расставить скобки по-другому? $A_{10*5} * (A_{5*100} * A_{100*50})$, сложность будет 5 * 100 * 50 + 10 * 5 * 50 = 25000 + 2500 = 27500. Ого, так это гораздо меньше первого результата. Оптимизация произошла из-за смены приоритета умножений.

Допустим, у нас 4 матрицы: ABCD. Сколькими способами можно расставить скобки? ((AB)C)D, A(B(CD)), (AB)(CD), (A(BC)D), A((BC)D) – пятью. Как понять, сколько вариантов может быть? Это можно узнать по числам Каталана (последовательность чисел, можно вычислить как C(n) = (2n)!/n!(n+1)!, вот начало последовательности: 1, 2, 5, 14, 42, 132, 429, 1430). Для 2-х матриц есть 1 вариант, для 3-х 2, для 4-х 5 и т. д. Очевидно, что считать всё это долго. И что делать?

Пусть у нас есть набор матриц $A_iA_{i+1}… A_{j-1}A_j$, и мы знаем, как расставить скобки: $(A_iA_{i+1}… A_k) (A_{k+1}…A_j)$. Матрицы внутри как-то перемножены. Если этот вариант оптимален $(A_iA_{i+1}… A_k) (A_{k+1}…A_j)$, то и эти оптимальны по-отдельности $A_iA_{i+1}… A_k$ и $A_{k+1}…A_j$. Если это не так, то при получении более оптимального ответа, в итоге мы получим более оптимальное решение. Нельзя, оптимизировав решение подзадачи, оптимизировать решение задачи – значит изначально решение задачи не было оптимальным.

Пусть число действий $A_iA_{i+1}… A_k = S_1$, а $A_{k+1}…A_j = S_2$, тогда итог будет $S_1 + S_2 + P_{i-1} * P_k * P_j$.

Что мы имеем в итоге? Чтобы найти расстановку скобок, для начала найдем расстановку скобок подпоследовательности, делящее её на две: $(A_iA_{i+1}… A_k) (A_{k+1}…A_j)$. То же самое делаем с подпоследовательностями.

То есть: $S(A_1… A_n) = \min(S(A_1… A_k) + S(A_{k+1}… A_n) + P_0 * P_k * P_n)$, минимум ищем по k от 1 до n-1.

Пример: $A_1A_2A_3A_4$, 4*3, 3*2, 2*5, 5*1.

С какой На какую	1	2	3	4
1	0	-	-	-
2	действие 1 4 * 3 * 2 = 24	0	-	-
3	действие 4 min(24 + 0 + 4 * 2 * 5, 0 + 30 + 4 * 3 * 5) = 64	действие 2 3 * 2 * 5 = 30	0	-
4	действие 6 min(0 + 16 + 4 * 3 * 1, 24 + 10 + 4 * 2 * 1, 64 + 1 + 4 * 5* 1) = 28	действие 5 min(30 + 0 + 3 * 5 * 1, 0 + 10 + 3 * 2 * 1) = 16	действие 3 2 * 5 * 1 = 10	0

Сложность $O(n^3)$ (все матрицы длины 2, потом 3 и т. д. + стрелочки, у нас будет три вложенных цикла, сначала от 2 до n, потом от 1 до n-l+1)

4. Задача о наибольшей общей подпоследовательности. Расстояние Левенштейна

Ключ: $LCS(X_1X_2...X_n, Y_1Y_2...Y_m) = \{$

при $X_n = Y_m$ $1+ LCS(X_1X_2...X_{n-1}, Y_1Y_2...Y_{m-1})$,

при $X_n \neq Y_m$ $\max(LCS(X, Y_1Y_2...Y_{m-1}), LCS(X_1X_2...X_{n-1}, Y))$.

при $m = 0 \,||\, n = 0$ $0 \}$

Таблица: если =, то диагональ + 1, если нет – то максимум из верхней и нижней. O(n*m).

Замена, вставка, удаление. $LD(X_1X_2...X_n, Y_1Y_2...Y_m) = \{$

при $X_n = Y_m$ $LD(X_1X_2...X_{n-1}, Y_1Y_2...Y_{m-1})$,

при $X_n \neq Y_m$ $\min(LD(X, Y_1Y_2...Y_{m-1})+insert, LD(X_1X_2...X_{n-1}, Y)+del, LD(X_1X_2...X_{n-1}, Y_1Y_2...Y_{m-1})+repl)$.

при $n = m = 0$ 0

при $m = 0$ delete * n

при $n = 0$ insert * m }

Таблица: если =, то диагональ. Если нет, то либо влево + удаление, либо вверх + вставка, либо диагональ + замена.

LCS – Longest Common Subsequence (наибольшая общая подпоследовательность)

Подпоследовательность – когда буквы идут слева направо, но не обязательно подряд (ABACACC - BCAC)

Задача о наибольшей общей подпоследовательности.

Дано:

S_1 = ABACABA, S_2 = BABCABBA, $|S_1| = n$, $|S_2| = m$.

Найти:

Максимальную длину общей подпоследовательности.

Решение:

Запишем условие и предполагаемое решение задачи в общем виде:

$X = X_1X_2...X_n$ – строка 1

$Y = Y_1Y_2...Y_m$ – строка 2

$R = R_1R_2...R_k$ – результат

Нам необходимо найти $R = LCS(X_1X_2...X_n, Y_1Y_2...Y_m)$

Рассмотрим X_n и Y_m. Они могут совпадать или не совпадать.

1) $X_n = Y_m$

   => тогда $X_n = Y_m = R_k$, эта буква – точно часть ответа, мы можем её отрезать и считать наибольшую подпоследовательность без неё, а потом просто включить.

   Отсюда $R_1R_2...R_{k-1} = LCS(X_1X_2...X_{n-1}, Y_1Y_2...Y_{m-1})$

2) $X_n \neq Y_m$
   В такой ситуации мы можем «отрезать» либо X_n, либо Y_m.
   Тогда $R = \max(LCS(X, Y_1Y_2...Y_{m-1}), LCS(X_1X_2...X_{n-1}, Y))$.
   2.1) Если $X_n \neq R_k$, то выкидываем X_n и $LCS(X_1X_2...X_{n-1}, Y)$
   2.2) Если $Y_m \neq R_k$, то выкидываем Y_m и $LCS(X, Y_1Y_2...Y_{m-1})$
   2.3) Если $Y_m \neq R_k$ и $X_n \neq R_k$, то $LCS(X_1X_2...X_{n-1}, Y_1Y_2...Y_{m-1})$

Как решать эту задачу с помощью ДМ?

Делим задачу на элементарные: будем считать LCS для элементарных случаев. Составим таблицу øABACABA × øBABCABBA:

	j =	0	1	2	3	4	5	6	7
i =		ø	A	B	A	C	A	B	A
0	ø	0	0	0	0	0	0	0	0
1	B	0	0	1	1	1	1	1	1
2	A	0	1	1	2	2	2	2	2
3	B	0	1	1	2	2	2	3	3
4	C	0	1	2	2	3	3	3	3
5	A	0	1	2	3	3	4	4	4
6	B	0	1	2	3	3	4	5	5
7	B	0	1	2	3	3	4	5	5
8	A	0	1	2	3	3	4	5	6

1. Самая элементарная задача – LCS пустоты и символа – равняется 0.
2. Начинаем сравнивать элементы (1,1). A ≠ B. Значит, надо «отрезать» либо X_i, либо Y_j. Идём вверх и вниз: смотрим значения ячеек (1-1, 1) = 0 и (1, 1-1) = 0 и выбираем из них максимум – 0.
3. Далее сравниваем элементы (1, 2). B = B. Смотрим значение ячейки по диагонали: (1-1, 2-1) = (0, 1). Смотрим значение – 0. 0 + 1 = 1.
4. Начинаем сравнивать элементы (1,3). A ≠ B. Значит, надо «отрезать» либо X_i, либо Y_j. Идём вверх и вниз: смотрим значения ячеек (1-1, 3) = 0 и (1, 3-1) = 1 и выбираем из них максимум – 1.
5. Аналогично для ячеек (1,4), (1,5).
6. Далее сравниваем элементы (1, 6). B = B. Смотрим значение ячейки по диагонали: (1-1, 6-1) = (0, 5). Смотрим значение – 0. 0 + 1 = 1.
7. Аналогичная схема для ячеек (2, 1), (2,2).
8. Далее сравниваем элементы (2, 3). B = B. Смотрим значение ячейки по диагонали: (2-1, 3-1) = (1, 2). Смотрим значение – 1. 1 + 1 = 2.

9 … n*m И ТАК ДАЛЕЕ

Нас интересует ответ - чёрная итоговая ячейка.

Чтобы восстановить путь, идём обратно, сравниваем буквы, если совпали – возврат по диагонали, если нет – выбираем максимум из лево и верха и идём туда. Когда пришли в ноль – стоп.

Рекуррентное соотношение для LCS:

$$LCS(X_1X_2\ldots X_n, Y_1Y_2\ldots Y_m) = \{$$

при $X_n = Y_m$	$1+ LCS(X_1X_2\ldots X_{n-1}, Y_1Y_2\ldots Y_{m-1})$,
при $X_n \neq Y_m$	$\max(LCS(X, Y_1Y_2\ldots Y_{m-1}), LCS(X_1X_2\ldots X_{n-1}, Y))$.
при m = 0 || n = 0	0

}

Сложность будет O((n+1)*(m+1)) = O(m*n).

Задача о расстоянии Левенштейна.

Дано:

Есть некое слово S_1 и слово S_2.

И элементарные преобразования (операции):

1. Замена (меняем ровно одну букву)
   МУХА -> МУКА
2. Вставка (вставляем одну букву на любую позицию слова)
   УХА -> МУХА

3. Удаление (удаляем одну любую букву)
   МУХА -> УХА

Найти:

С помощью какого минимального количества операций (с учётом их стоимости) можно получить из слова S_1 слово S_2?

Решение:

Можно ли решить задачу методом ДМ? Можно. Почему? Потому что это задача оптимизации.

Задача очень похожа на поиск общей подстроки (ЛК 1). LD – расстояние Левенштейна.

delete, insert, replace – стоимости этих операций.

Рассмотрим X_n и Y_m. Они могут совпадать или не совпадать.

1. $X_n = Y_m$

   => $LD(X_1X_2…X_n, Y_1Y_2…Y_m) = LD(X_1X_2…X_{n-1}, Y_1Y_2…Y_{m-1})$

   LD(МУХА, КОЛА) = LD(МУХ, КОЛ)

   Если последние совпавшие буквы убрать, задача, по сути, не поменяется.

2. $X_n \neq Y_m$
   Выбираем лучший по стоимости вариант из следующих трёх:
   2.1. $LD(X, Y) = LD(X_1X_2…X_{n-1}, Y)$ + delete
   LD(МУХА, СЛОН) = LD(МУХ, СЛОН)
   2.2. $LD(X, Y) = LD(X, Y_1Y_2…Y_{m-1})$ + insert
   LD(МУХА, СЛОН) = LD(МУХАН, СЛОН)
   2.3. $LD(X, Y) = LD(X_1X_2…X_{n-1}, Y_1Y_2…Y_{m-1})$ + replace
   LD(МУХА, СЛОН) = LD(МУХН, СЛОН)

   min(2.1, 2.2, 2.3)

Чертим матрицу. Если буквы совпали, идём по диагонали («отсекаем» совпавшую часть). Если нет, то либо идём влево (+ удаление), либо идём вверх (+ вставка) и плюс к этим операциям можем идти из той ячейки, куда пришли, по диагонали (+ замена).

Расстояние Левенштейна в рекуррентном соотношении:

$LD(X_1X_2…X_n, Y_1Y_2…Y_m) = \{$
при $X_n = Y_m$ $LD(X_1X_2…X_{n-1}, Y_1Y_2…Y_{m-1})$,
при $X_n \neq Y_m$ $\min(LD(X, Y_1Y_2…Y_{m-1})+insert, LD(X_1X_2…X_{n-1}, Y)+del, LD(X_1X_2…X_{n-1}, Y_1Y_2…Y_{m-1})+repl)$.
при n = m = 0 0
при m = 0 delete * n
при n = 0 insert * m
}

Пример: AAB -> BACA, d (delete) = 3, i (insert) = 2, r (replace) = 4.

	j	0	1	2	3
k		Ø	A	A	B
0	Ø	0	3	6	9
1	B	2	4	7	6
2	A	4	2	4	7
3	C	6	4	6	8
4	A	8	6	4	7

1. Смотрим первую строку. Как из A получить ∅? Удалить. Стоимость d = 3.
2. Как из AA получить ∅? Удалить дважды. Стоимость 2 * d = 2 * 3 = 6.
3. Как из AAB получить ∅? Удалить трижды. Стоимость 3 * d = 3 * 3 = 9.
4. Смотрим первый столбец. Как из ∅ получить B? Вставить B. Стоимость i = 2.
5. Как из ∅ получить BA? Вставить B, вставить A. Стоимость 2 * i = 2 * 2 = 4.
6. И т д
7. Смотрим ячейку (1,1). A ≠ B. У нас есть три варианта:
   7.1. Получить из первого слова без последней буквы второе слово без последней буквы и заменить. То есть A->∅, B->∅, можно сделать замену A->B. Замена r = 4. Так как стоимость (0, 0) = 0, то итог 0 + 4 = 4.
   7.2. Отрезаем у второго слова последнюю букву и вставляем букву. Делаем шаг вверх. B->∅, можно выполнить вставку ∅->B. Вставка i = 2. Так как стоимость (0, 1) = 3, то итог 3 + 2 = 5.
   7.3. Отрезаем у первого слова последнюю букву и удаляем букву. Делаем шаг влево. B->∅, можно выполнить удаление ∅->B. Удаление d = 2. Так как стоимость (1, 0) = 2, то итог 2 + 3 = 5.

   Выбираем из трёх этих вариантов минимум min(4, 5, 5) = 4. Пишем в клетку 4.
8. Далее аналогично:
   Если буквы равны, то стоимость та же, что и по диагонали.
   Если буквы не равны, то стоимость вверх и влево – 5, по диагонали – 4 и + значение ячейки.

Нас интересует финальная красная клетка. Чтобы восстановить порядок действий, идём обратно.

Оптимизировать память, если не нужно по задаче хранить все значения, можно решать только через запоминание текущей и предыдущей строки.

5. Жадные алгоритмы. Основные этапы построения решения, отличие от динамического программирования.

Жадный алгоритм -

алгоритм, который будет давать **оптимальный ответ не в целом**, а на каком-то **текущем шаге**. Мы придумаем простое **правило**, которое позволит **максимизировать ответ на каждом следующем** шаге. И этот ответ будет разбивать задачу так, чтобы все подзадачи, кроме одной, были пустыми.

Отличие от ДП:

В ДП мы решаем все задачи и по каким-то признакам выбираем одну. В ЖА мы делим задачу на одну подзадачу, то есть на каждом шаге решение оптимальное.

Замечание:

Алгоритма, когда применяется жадный алгоритм, нет. ЖА периодически используется для случаев, когда не нужен точный ответ, а близкий к нему (на больших областях).

6. Задачи о дискретном и непрерывном рюкзаках.

Ключ: ценность, сортируем по убыванию, дискретная не решается с помощью ЖА, контрпример.

Любую ли задачу из ДМ можно решить жадным алгоритмом? Нет. Например, дискретную версию задачи про рюкзак.

Задачи о рюкзаках (дискретная – только целый предмет можно взять, есть непрерывная – там можно брать доли предметов).

Дано:

Есть рюкзак, который можно нагружать каким-то максимальным весом w. И есть предметы, которые можно складывать в него (параметры: вес(w_k)-стоимость(p_k)).

Допустим, w_1=10, w_2=20, w_3=30, p_1=60, p_2=100, p_3=120;

Вопрос:

Нужно найти, какие предметы можно получить для получения максимальной стоимости при условии, что их вес не должен превышать допустимый для рюкзака.

Решение:

Введём понятие ценности: $c_i = p_i/w_i$; $c_1 = 6$, $c_2 = 5$, $c_2 = 4$.

Сортируем вещи в порядке убывания по ценности.

Решение для непрерывного рюкзака:

Смотрим: можем полностью положить первый предмет? Можем, кладём. Можем второй положить? Можем, кладём. Третий можем положить? Весь – нет. Выбираем часть так, чтобы поместилось. Получаем ответ. 10+20+10 = 40.

Решение для дискретного рюкзака:

Если действовать жадным алгоритмом в этом случае, то получим стоимость 60+100 = 160, хотя правильный ответ 100+120 = 220. Жадный алгоритм тут не работает.

Проще всего доказать, что задача не решается жадным алгоритмом – контрпример.

7. Задача об оптимальном выборе процессов.

Ключ: В лоб: M – множество игр в промежутке от начала первой до конца второй, M' – подмножество M, множество непересекающихся игр, максимальное количество элементов – ответ.
ДП: какая-то игра гарантированно входит, тогда выбираем максимум из оптимального ответа до неё, после и неё – $O(n^3)$.
ЖА: сортируем по концу - неубывание. Первый процесс – текущий, он входит в ответ, потом процессы, которые начинаются раньше, чем первый кончился, пропускаем. Первый процесс, начинающийся позже, чем конец текущего, назовём текущим - O (n * сортировка).

Дано:

Есть набор матчей в доте, которые сыграны, но результаты их неизвестны. Матчи ведутся по разным стримам. Есть расписание стримов (время начала и конца игры).
Допустим, есть временные отрезки, их количество - n:
A [3, 6)
B [2, 5)
C [1, 2)
D [7, 12)
E [8, 10)
F [10, 12)

Найти:

Наибольшее количество игр, которое мы можем посмотреть полностью (продолжительность не важна).
Заметка: последовательности (A, D), (B, E, F) корректны, (D, E) - нет.

Решение 1 – в лоб:

Самое простое решение – в лоб, перебор. Однако сложность будет O (2^n * n^2) – перебор * проверку отсутствия пересечений.

Оптимизировать решение можно через сортировку отрезков. Отсортируем игры по началу и переименуем их (для удобства). Также создадим игры A_1 и A_8 – это самая ранняя и самая поздняя игра, то есть ими задаём границы времени.

i	1	2	3	4	5	6	7	8
	X	C	B	A	D	E	F	Y
	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8
S_i	$-\infty$	1	2	3	7	8	10	$+\infty$
F_i	0	2	5	6	12	10	12	$+\infty$

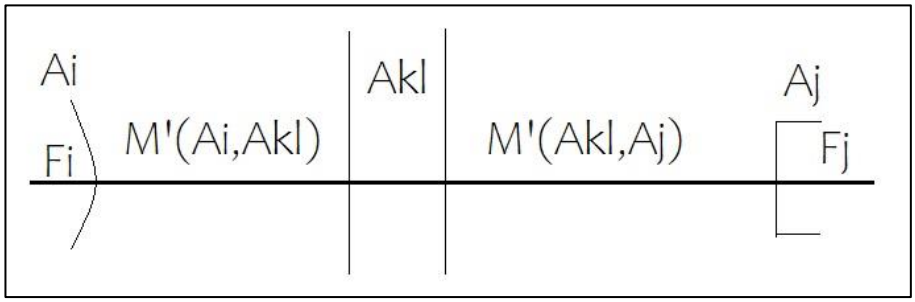
Пусть $M(A_i, A_j) = \{A_k \mid S_k >= F_i, F_k <= S_j\}$

Тогда $M(A_2, A_7) = \{A_k \mid S_k >= 2, F_k <= 10\}$ – то есть игры, которые начинаются позже двух, заканчиваются раньше 12-ти. Тогда k = 3, 4, 6. Однако мы видим, что A_3, A_4 пересекаются, а это противоречит условию. Что мы делаем? Исключаем пересечения.

$M'(A_i, A_j)$ – подмножество $M(A_i, A_j)$, такое, что для всех A_k, A_l $S_k >= F_l$, $F_k <= S_l$. Тогда $M'(A_2, A_7) = \{A_3\}$, $\{A_6\}$, $\{A_3, A_6\}$, $\{A_4, A_6\}$. Максимальное количество элементов в подмножестве – 2.

Решение 2 - ДП:

Пусть A_{kl} – игра, которая гарантированно входит. Тогда мы можем разбить игры на две группы (до неё включительно) и после: $M'(A_i, A_j) = M'(A_i, A_{kl}), M'(A_{kl}, A_j), A_{kl}$.



Дальше ищем какой-то оптимальный ответ среди $M'(A_i, A_{kl})$, $M'(A_{kl}, A_j)$, и выбираем максимум из $M'(A_i, A_{kl})$, $M'(A_{kl}, A_j)$, A_{kl}.

Мы можем посчитать $M'(A_1, A_2) = M'(A_2, A_3) = M'(A_3, A_4) = 0$.
Потом $M'(A_1, A_3) = M'(A_1, A_2) + M'(A_2, A_3) + 1$

$M'(A_1, A_4) = \max(M'(A_1, A_2) + 1 + M'(A_2, A_4), M'(A_1, A_3) + 1 + M'(A_3, A_4))$ и т д.

Сложность кубическая

Решение 3 - ЖА:

Пусть есть $M(i, j) = \{b_1, b_2, \dots b_n\}$,

$M'(i, j) = M'(i, b_1) + 1 + M'(b_1, j)$, при этом $M'(i, b_1)$ всегда будет 0, то есть $M'(i, j) = 1 + M'(b_1, j)$.

Мы бьём подзадачу на одну.

Что означает пусть $M'(i, b_1)$ всегда будет 0? Это значит, что процесс b_1 будет заканчиваться раньше, чем все остальные.

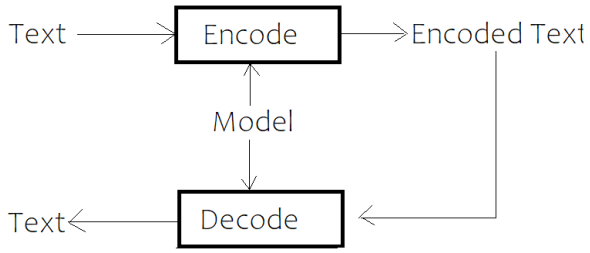
Мы отсортировали все процессы по времени **их окончания по неубыванию**. Мы берём самый первый процесс, называем его текущим и говорим, что он входит в ответ, потом процессы, которые начинаются раньше, чем первый кончился, пропускаем. Первый процесс, который начинается позже, чем конец текущего, назовём текущим.

Сложность O(nlogn), n – проход, logn – сортировка.

8. Коды Хаффмана. Построение, доказательство корректности. Построение за линейное время

Ключ: вариант за nlogn: сортируем по частоте в кучу, вытаскиваем два минимальных значения (каждый перекинули в конец, отсортировали), из вытащенных элементов строим узел дерева. Кладём узел в кучу. Потом вытащили ещё два элемента и тп. Частота суммируется. Обход: лево – 0, право – 1. Декодируем так же. На каждом шаге берём минимум – ЖА.
За n: два массива сорт по возрастанию, минимум суммы двух редких из: первого, второго и первого, кладём во второй массив. По итогу получим дерево как в 1. Вычёркиваем по 2 элемента – O(n).

Выкладка про кодирование:



Суть кодирования: есть текст, мы его закодировали, получили зашифрованный текст. Мы можем его расшифровать (декодировать) и получить исходный текст.

Идея в чём: размер закодированного текста должен быть меньше, чем исходный, чтобы было оптимальнее хранить.

Пример (самый простой): есть текст “ABC”, пусть кодирование имеет вид: код ascii буквы – 65 (То есть A будет не 65, а 0, B не 66, а 1 и т д). Закодированный текст будет выглядеть так: «123». Раскодируем его: символ в ascii на позиции 1 + 65 – A, 2 + 65 – B и т д. Получим текст «ABC». Он равен исходному.

Идея префиксного кода: ни один код не может быть префиксом другого (для устранения ситуаций по типу 00001 = ААК и 00001 = АБ).

Выкладка про heap (куча):

Что такое куча?

Есть массив, a_i имеет двух потомков a_{2i+1} и a_{2i+2}, такие, что $a_i \le a_{2i+1}$ и $a_i \le a_{2i+2}$ (или больше либо равно)

Как сортируется и строится?

Берём какой-то массив и, начиная с середины запускаем процедуру keepify.

Что такое keepify?

Берём элемент с индексом i, дальше смотрим элементы с индексами 2*i + 1 и 2*i + 2, если i минимальный, то всё хорошо, переходим к i – 1 и делаем для него всё то же самое, если не минимальный, то меняем его местами с элементом и дальше от того элемента, который мы поставили на i-ю позицию, снова начинаем keepify.

Пример сортировки:

7	12	1	3	6	3	4	5	0	2
0	1	2	3	4	5	6	7	8	9

min (6, 2, ‘\n’) = 2.
Позиции, мы бы перешли на 8 позиции, но детей у неё нет. Поэтому new_i = 4 – 1 = 3

7	12	1	3	2	3	4	5	0	6

0	1	2	3	4	5	6	7	8	9

min (3, 5, 0) = 0

Позиции 8*2+1 нет, new_i = 3 – 1 = 2

7	12	1	0	2	3	4	5	3	6
0	1	2	3	4	5	6	7	8	9

min (1, 3, 4) = 1

7	12	1	0	2	3	4	5	3	6
0	1	2	3	4	5	6	7	8	9

min (12, 3, 2) = 2.

Позиция 4*2+1 есть. Поэтому new_i = i(min) = 4.

7	2	1	0	12	3	4	5	3	6
0	1	2	3	4	5	6	7	8	9

min (12, 6, ‘\n’) = 6.

7	2	1	0	6	3	4	5	3	12
0	1	2	3	4	5	6	7	8	9

7	2	1	0	6	3	4	5	3	12
0	1	2	3	4	5	6	7	8	9

И так далее.

Алгоритм кодирования по Хаффману за nlogn:

Есть слово: АБРАКАДАРБА.

Применим для него код Хаффмана, чтобы найти оптимальное построение для каждой буковки.

Кодирование.

Посчитаем, сколько раз встречается каждая буква. Запихнём буквы в heap.

Почему куча? Можно быстро вытащить.

Буква	А	Б	Д	К	Р
Частота	5	2	1	1	2

Далее отсортируем буквы по частоте их встречаемости в коде (от меньшего к большему).

В куче это будет выглядеть так:

Буква	Д	К	А	Б	Р
Частота	1	1	5	2	2

1) Берём букву Д и удаляем её из heap (как это сделать: меняем Р и Д местами, Д удаляем, Р кучу сортируем).
2) Потом берём следующий минимальный элемент – К. Удаляем его из heap.

Буква	Б	Р	А
Частота	2	2	5

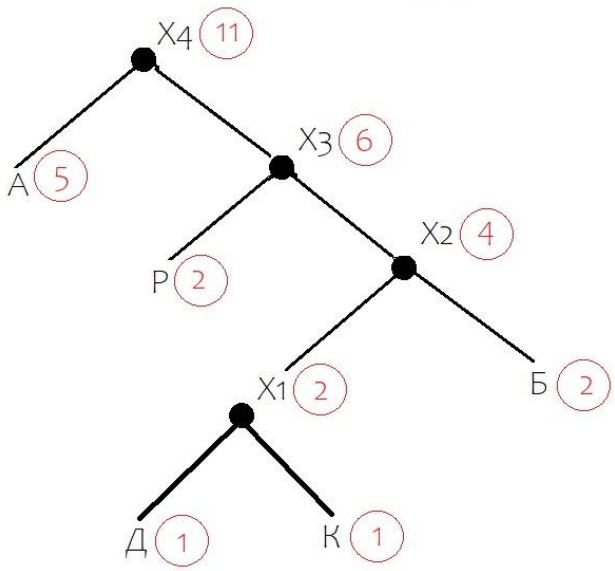
3) Строим дерево, корень – некая вершина X_1, её дети – левый Д и правый К. Частота корня – частота сумм двух детей (то есть 2).
   Что это всё значит? Наше слово «АБРАКАДАБРА» будет иметь вид «АБРАX_1АX_1АБРА»
4) Кладём X_1 в heap (на место К) и перестраиваем его, если надо (не надо сейчас).

Буква	X_1	Б	Р	А
Частота	2	2	2	5

5) Снова извлекаем и удаляем два минимума: X_1 и Б, строим вершину X_2, её дети – левый X_1 и правый Б. Частота корня – частота сумм двух детей (то есть 2 + 2 = 4).
6) Кладём X_2 в heap (на место Б) и перестраиваем его.

Буква	Р	X_2	А
Частота	2	4	5

И так далее. По итогу получим:



Что мы делаем дальше? Как мы считаем коды Хаффмана?

Обойдём это дерево. Если мы идём налево, дописываем 0, направо – 1.

То есть:

А -> 0
Р -> 10
Б -> 111
Д -> 1100
К -> 1101

АБРАКАДАБРА -> 01111001101011000111100

Это непрефиксный код.

Сложность: пирамида строиться за nlog(n), извлечение минимума – log(n), вставка как построение, вершин у нас O(2n-1) = O(n), поэтому код строится за n. По итогу nlogn.

Теперь декодирование.

01111001101011000111100

1) Считали ноль. Смотрим: это буква «А». Пишем в ответ.
2) Дальше считываем. 1. Смотрим: это узел «X_3». Узел – это не лист. Считываем дальше – 1. Это узел «X_2». Узел – это не лист. Считываем дальше – 1. Это буква «Б». Пишем в ответ.
3) И т д

Почему этот алгоритм жадный? На каждом шаге мы берём минимум – самый оптимальный вариант.

Алгоритм кодирования по Хаффману за n:

Заведём два массива.

Первый: отсортируем буквы в порядке возрастания.

Буква	Д	К	Б	Р	А
Частота	1	1	2	2	5

Второй: пока пустой.

Так как оба массива отсортированы по возрастанию, делаем вот что: возьмём три пары:

1) Две самые редко встречаемые буквы из первого массива.
2) Одна самая редкая из второго массива и одну самая редкая из первого.
3) Две самые редко встречаемые буквы из второго массива.

Суммируем частоты в паре, определяем, какая частота минимально возможная из трёх вариантов. Те буквы, чьи частоты минимально возможные, их вычёркиваем. И в конец второго массива добавляем их сумму.

То есть:

1. min (Д+К, ø, ø) = 2.

Буква	Б	Р	А
Частота	2	2	5

X_1
2

2. min (Б+Р, X_1+Б, ø) = min (4, 4, ø) = 4. Выберем, допустим, второй вариант.

Буква	Р	А
Частота	2	5

X_2
4

3. min (Р+А, X_2+Р, ø) = min (7, 6, ø) = 6.

Буква	А
Частота	5

X_3
6

4. min (ø, А+X_3, ø) = 11.

X_4
11

Получим то же самое дерево, что и при логарифмическом решении.

Сложность: два массива содержат 2n-1 элементов, каждый шаг вычёркиваем два элемента из массивов, добавляем один, всего n шагов получается, поэтому сложность линейная O(n).

что-то есть!!!!!!!!!!!!!!!!!, вроде декодирование за линейное время

9. Задача о наибольшей возрастающей подпоследовательности, ее использование для решения задачи о наибольшей общей подпоследовательности

Ключ: LIS: покрытие, если > + новый список, <= кладём в максимально близкий список. Кол-во списков - ответ. ЖА, тк всегда невозрастание в списках. Восстановить – ссылка на последнее число. O(n * поиск= n, бинарный logn).

Позиции букв 1 отсортируем, подставим последовательность вместо букв в 2, запустим LIS, кол-во списков – ответ, по ссылкам – индексы неотсортированные из 1. R*logR, длина 2 в цифрах, для длинных строк - ок

Задача LIS через жадный алгоритм.

Дано:

LIS – longest increase subsequence. Есть строка чисел, из этой строки надо вытащить строку, в которой числа строго возрастающие и при этом длина этой подстроки должна быть наибольшей возможной. Например:

LIS (7 15 34 2 26 9 31) = 4.

Найти:

Алгоритм решения этой задачи?

Решение:

Как это сделать: идём по списку и каждый раз берём элемент, который и мы пытаемся добавить его в какой-то список.

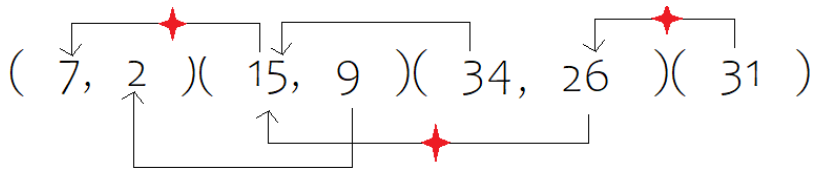
Первый элемент: 7, создаём новый список, кладём 7. (7)

Второй элемент: 15. Он меньше либо равен, чем 7? Нет. Создаём второй список, кладём туда 15. (7)(15).

Третий элемент: 34. Он меньше либо равен, чем последний элемент первого списка - 7? Нет. Он меньше либо равен, чем последний элемент второго списка - 15? Нет. (7)(15)(34).

Четвёртый элемент: 2. Он меньше либо равен, чем последний элемент первого списка - 7? Да. Кладём в список. (7, 2)(15)(34). Пятый элемент: 26. (7, 2)(15)(34, 26).

И так далее. Итог: (7, 2)(15, 9)(34, 26)(31). Эта штука – покрытие. Длина самой большой подпоследовательности – количество списков. Как восстановить нашу последовательность? При вставке делаем ссылку на текущее последнее число в предыдущем списке.



Ответ: 7, 15, 26, 31

Берём любое число из последнего списка и «идём» по нему.

Другой пример. Пусть покрытие будет (3, 2, 1, 0)(6, 4, 3, 2)(5, 5)(7). Ссылка из 4 была в 1. То есть итоговый ответ 7 5 4 1. Однако, мы можем для ответа взять элементы выше первее 1 и меньше 4, они будут подходить. То есть: 7 5 4 2, 7 5 4 3 – тоже правильные ответы.

Это жадный алгоритм – всегда невозрастание в списках смотрим. Сложность в худшем случае O(n^2).

Оптимизируем. Можем за счёт бинарного поиска уменьшить создание списка за O(nlogn), так как все элементы внутри списка невозрастающие.

Задача LCS через жадный алгоритм (LIS).

Дано:

S_1 = ABACABA, S_2 = BABCABBA, | S_1| = n, | S_2 | = m.

Найти:

Максимальную длину общей подпоследовательности.

Решение:

Выпишем буквы и индексы, на которых они стоят:

A: 0, 2, 4, 6.
B: 1, 5.
C: 3.

Отсортируем в порядке убывания (готовим для LIS, чтобы одну и ту же букву нельзя было брать):

A: 6, 4, 2, 0.
B: 5, 1.
C: 3.

Далее берём вторую строку и заменяем её на индексы первой строки.

S_2 = BABCABBA = (5, 1)(6, 4, 2, 0)(5, 1)(3)(6, 4, 2, 0)(5, 1)(5, 1)(6, 4, 2, 0) =

= 5, 1, 6, 4, 2, 0, 5, 1, 3, 6, 4, 2, 0, 5, 1, 5, 1, 6, 4, 2, 0.

Решаем задачу о LIS (наибольшей возрастающей подпоследовательности).

(5, 1, 0, 0, 0)(6, 4, 2, 1, 1, 1)(5, 3, 2)(6, 4, 4)(5, 5)(6). Списков 6 – это и есть искомый ответ. А как определить строку? Есть пройти по ссылкам, то получим 0 1 3 4 5 6. Это индексы из первой строки. То есть ABCABA.

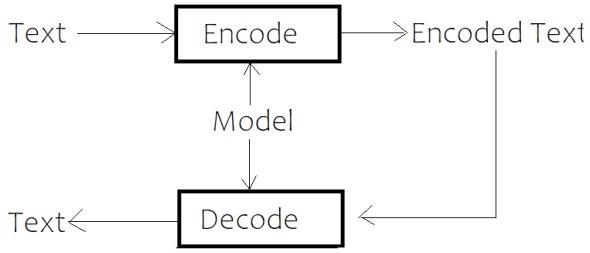
Сложность O(R*logR), R – длина преобразованной S_2. Для длинных строк работает хорошо (быстрее, чем m*n).

Блок II. Сжатие текстов

10. Постановка задачи сжатия текста. Сжатие с потерями и без потерь. Словарные и символьные алгоритмы сжатия текста.

Ключ: процесс, вход, модель, другой объект, исходник. Символ, слово, оптимально хранить.

Сжатие текста – это процесс, который на вход получает объект, с помощью модели и кодирования даёт другой объект, который с использованием той же модели и декодирования позволяет получить исходный объект.



Идея в чём: размер закодированного текста должен быть меньше, чем исходный, чтобы было оптимальнее хранить.

По считыванию объекта из исходной базы алгоритмы сжатия делятся на:

1. Символьные
   Есть входная строка: abc. a соответствует код 01, b соответствует 011, c - 0111. У каждого символа есть свой код. Пример: код Хаффмана, арифметическое кодирование.
2. Словарные
   Для каждого слова есть свой код. Не для символа, для слова! Пример: LZ-77, LZ-78.

11. Модели сжатия. Контекстно-ограниченная модель, модель конечного автомата.

Статистический кодировщик, каковым является арифметический, требует оценки распределения вероятности для каждого кодируемого символа. Проще всего присвоить каждому символу постоянную вероятность, независимо от его положения в тексте, что создает простую контекстуально-свободную модель. Например, в английском языке вероятности символов ".", "e", "t" и "k" обычно составляют 18%, 10%, 8% и 0.5% соответственно (символ "." используется для обозначения пробелов). Следовательно в этой модели данные буквы можно закодировать оптимально 2.47, 3.32, 3.64 и 7.62 битами с помощью арифметического кодирования. В такой модели каждый символ будет представлен в среднем 4.5 битами. Это является значением энтропии модели, основанной на вероятности распределения букв в английском тексте. Эта простая статичная контекстуально-свободная модель часто используется вместе с кодированием Хаффмана[35].

Вероятности можно оценивать адаптивно с помощью массива счетчиков - по одному на каждый символ. Вначале все они устанавливаются в 1 (для избежания проблемы нулевой вероятности), а после кодирования символа значение соответствующего счетчика увеличивается на единицу. Аналогично, при декодировании соответствующего символа раскодировщик увеличивает значение счетчика. Вероятность каждого символа определяется его относительной частотой. Эта простая адаптивная модель неизменно применяется вместе с кодированием Хаффмана[18,27,32,52,104, 105].

Более сложный путь вычисления вероятностей символов лежит через определение их зависимости от предыдущего символа. Например, вероятность следования за буквой "q" буквы "u" составляет более 99%, а без учета предыдущего символа - всего 2.4%(2). С учетом контекста символ "u" кодируется 0.014 бита и 5.38 бита в противном случае. Вероятность появления буквы "h" составляет 31%, если текущим символом

является "t", и 4.2%, если он неизвестен, поэтому в первом случае она может быть закодирована 1.69 бита, а во втором - 4.6 бита. При использовании информации о предшествующих символах, средняя длина кода (энтропия) составляет 3.6 бита/символ по сравнению с 4.5 бита/символ в простых моделях.

Этот тип моделей можно обобщить относительно о предшествующих символов, используемых для определения вероятности следующего символа. Это определяет контекстно-ограниченную модель степени о. Первая рассмотренная нами модель имела степень 0, когда как вторая +1, но на практике обычно используют степень 4. Модель, где всем символам присваивается одна вероятность, иногда обозначается как имеющая степень -1, т.е. более примитивная, чем модель степени 0.

Контекстно-ограниченные модели неизменно применяются адаптивно, поскольку они обладают возможностью приспосабливаться к особенностям сжимаемого текста. Оценки вероятности в этом случае представляют собой просто счетчики частот, формируемые на основе уже просмотренного текста.

Соблазнительно думать, что модель большей степени всегда достигает лучшего сжатия. Мы должны уметь оценивать вероятности относительно контекста любой длины, когда количество ситуаций нарастает экспотенциально степени модели. Т.о. для обработки больших образцов текста требуется много памяти. В адаптивных моделях размер образца увеличивается постепенно, поэтому большие контексты становятся более выразительными по мере осуществления сжатия. Для оптимального выбора - большого контекста при хорошем сжатии и маленького контекста при недостаточности образца - следует примененять смешанную стратегию, где оценки вероятностей, сделанные на основании контекстов разных длин, объединяются в одну общую вероятность. Существует несколько способов выполнять перемешивание. Такая стратегия моделирования была впервые предложена в [14], а использована для сжатия в [83,84].

Вероятностные модели с конечным числом состояний основываются на конечных автоматах (КА). Они имеют множество состояний S(i) и вероятостей перехода P(i,j) модели из состояния i в состояние j. При этом каждый переход обозначается уникальным символом. Т.о., через последовательность таких символов любой исходный текст задает уникальный путь в модели (если он существует). Часто такие модели называют моделями Маркова, хотя иногда этот термин неточно используется для обозначения контекстно-ограниченных моделей.

Модели с конечным числом состояний способны имитировать контекстно-ограниченные модели. Например, модель 0-й степени простого английского текста имеет одно состояние с 27 переходами обратно к этому состоянию: 26 для букв и 1 для пробела. Модель 1-й степени имеет 27 состояний, каждое с 27 переходами. Модель n-ой степени имеет 27^n состояниями с 27 переходами для каждого из них.

Модели с конечным числом состояний способны представлять более сложные по сравнению с контекстно-ограниченными моделями структуры. Простейший пример дан на рисунке 1. Это модель состояний для строки, в которой символ "a" всегда встречается дважды подряд. Контекстуальная модель этого представить не может, поскольку для оценки вероятности символа, следующего за последовательностью букв "a", должны быть рассмотрены произвольно большие контексты.

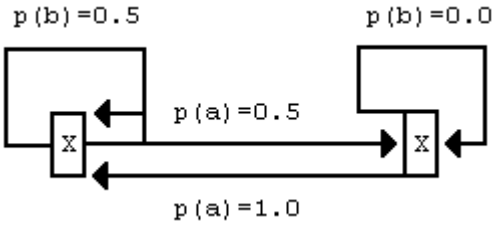


Рисунок 1. Модель с ограниченным числом состояний для пар "a".

Помимо осуществления лучшего сжатия, модели с конечным числом состояний быстрее в принципе. Текущее состояние может замещать вероятность распределения для кодирования, а следующее состояние просто определяется по дуге перехода. На практике состояния могут быть выполнены в виде связного списка, требующего ненамного больше вычислений.

К сожаления удовлетворительные методы для создания хороших моделей с конечным числом состояний на основании образцов строк еще не найдены. Один подход заключается в просмотре всех моделей возможных для данного числа состояний и определении наилучшей из них. Эта модель растет

экспотенциально количеству состояний и годится только для небольших текстов [30,31]. Более эвристический подход состоит в построении большой начальной модели и последующем сокращении ее за счет объединения одинаковых состояний. Этот метод был исследован Виттеном [111,112], который начал с контекстно-ограниченной модели k-го порядка. Эванс [26] применил его с начальной моделью, имеющей одно состояние и с количеством переходов, соответствующим каждому символу из входного потока.

12. Модели данных. Статическая модель, полустатическая модель, адаптивная модель. Проблема символов с нулевой частотой для адаптивных моделей.

Ключ: статическая – сначала кодировка, потом текст. Полустатическая – узнаём характеристики текста, кодируем (храним и код, и текст). Адаптивная – считали символ, закодировали, изменили модель.

По построению модели:

1. Статическая (С)
   Сначала есть кодировка для символов, потом проходимся по тексту.
2. Полустатическая (ПС).
   Берём текст и узнаём его характеристики, а после этого кодируем. От статической модели отличается тем, что статическая модель кодируется для всех текстов одинакова (А – всегда 1, В – всегда 2 и тп, хотя в тексте В может и не быть). Проблема в том, что нужно хранить и модель, и текст.
3. Адаптивная (А)
   Есть базовая модель, которая с новым символов каждый раз адаптируется к тексту, то есть изменяется.
   Однако порядок кодирования вот какой: мы считали символ, сначала закодировали объект, потом занесли букву в модель, изменяя её. В противном случае мы не сможем декодировать текст.

Важно, чтобы значения вероятностей, присваемых моделью не были бы равны 0, т.к. если символы кодируются -log p битами, то при близости вероятности к 0, длина кода стремится к бесконечности. Нулевая вероятность имеет место, если в образце текста символ не встретился ни разу - частая ситуация для адаптированных моделей на начальной стадии сжатия. Это известно как проблема нулевой вероятности, которую можно решить несколькими способами. Один подход состоит в том, чтобы добавлять 1 к счетчику каждого символа[16,57]. Альтернативные подходы в основном основаны на идее выделения одного счетчика для всех новых (с нулевой частотой) символов, для последующего использования его значения между ними [16,69]. Сравнение этих стратегий может быть найдено в [16,69]. Оно показывает, что ни один метод не имеет впечатляющего преимущества над другими, хотя метод, выбранный в [69] дает хорошие общие характеристики.

13. Коды Хаффмана. Статическая, полустатическая и адаптивная модели.

Ключ: полустатическая – путь+код+путь… - модель, восстанавливаем – строим дерево по модели. Адаптивная – массив невозрастающий по частотам, добавляем искусственную вершину, битьё – предки + 1, дети в массив, если нарушилось условие – меняем детей. Декодируем аналогично.

Статический пример сжатия текста (символьный):

Текст: АБРАКАДАБРА.

Модель (непрефиксная – нет ни одного кода, который является префиксом другого. Можем дерево построить):

Буква	Код
А	000
Б	11100
Д	11101
Р	0011
К	0010

Тогда закодированный текст:

00011100001100000100001110100011100001100 0

Декодируем:

Буква с кодом 0 есть? Нет. 00? Нет. 000? Да. +А. 1 есть? Нет и тп.

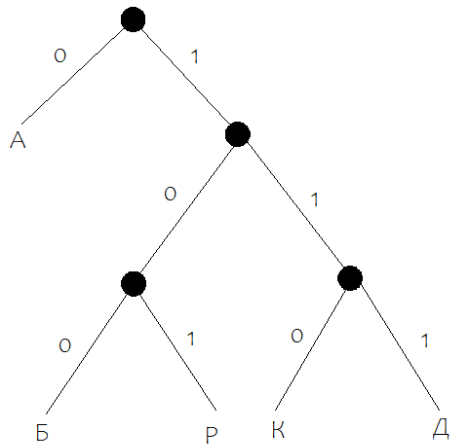
АБРАКАДАБРА

Полустатический пример сжатия текста (символьный, по Хаффману):

Текст: АБРАКАДАБРА.

Кодировка (вопрос 8):

Буква	Код
А	0
Б	100
Д	101
Р	110
К	111



Сюрреализуем модель.

Начинаем с корня. Если мы сейчас в вершине, не в листе, пишем 0, потом идём в левое поддерево и так сюрреализуем его, потом в правое и то же самое. Если мы в листе – пишем 1 и код символа.

Как это всё работает.

Для этого дерева:

1) Мы в корне. Пишем ноль, идём в левое поддерево. Пишем 1 и код буквы А.
   01[код А].
2) Идём в правое поддерево. Там внутренняя вершина. Пишем 0.
   01[код А]0.
3) Мы во внутренней вершине, идём в левое поддерево. Опять встречаем внутреннюю вершину, пишем 0.
   01[код А]00.
4) Мы во внутренней вершине, идём в левое поддерево. Видим букву: Б. Пишем 1, код буквы.
   01[код А]001[код Б].
5) Идём в правое поддерево, там Р.
   01[код А]001[код Б]1[код Р].
6) И так далее. Получим:
   01[код А]001[код Б]1[код Р]01[код К]1[код Д] – это модель.

Как декодируем:

01[код А]001[код Б]1[код Р]01[код К]1[код Д].

Восстанавливаем дерево. 0 – внутренняя вершина. Дальше идёт лист – буква А. Далее внутренняя вершина, идём к предку. Там внутренняя вершина, рисуем. Дальше… И так далее.

Псевдокод для рекурсивного обхода:

```
Build (T):
    A = readbit()
    If A == 0:
        Build (T.left)
        Build (T.right)
    else:
        b = readcharacter()
        put(b)
        return
```

Адаптивные коды Хаффмана.

Идея:

Возьмём слово. На первом этапе заводим вершину с искусственным символом и частотой один. Далее считываем букву и, если наша буква присутствует в дереве Хаффмана, отправляем её код, если не присутствует – отправляем код искусственного символа, после чего делим искусственную вершину на две, то есть делаем её внутреннем. В левое поддерево отправится наша буква считанная, в правую – искусственный символ. Также есть правило для дерева: для всех детей слева направо должен быть порядок не возрастания, то есть массив из вершин должен быть невозрастающий.

Пример:

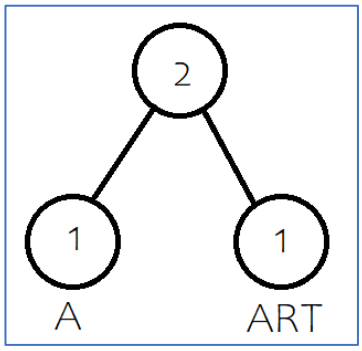
Дана строка «АБРАК».

Кодирование:

1) Создаём искусственную вершину с частотой встречаемости 1 и массив вершин.

Вершина	ART_0
idex	0
частота	1

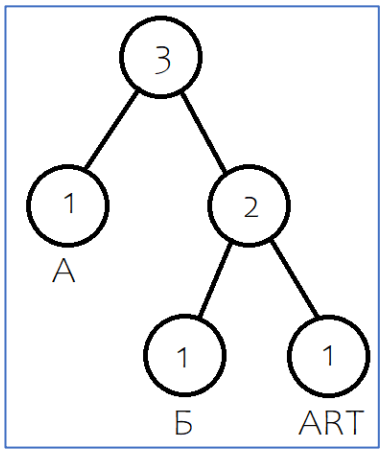
2) Считываем букву строки – А. Её в дереве нет, значит, надо отправить код искусственного символа – пустую строку и код А. В результат пишем код буквы А и бьём искусственную вершину.



Результат: code[A].

Вершина	ART_0	A	ART
idex	0	1	2
частота	2	1	1

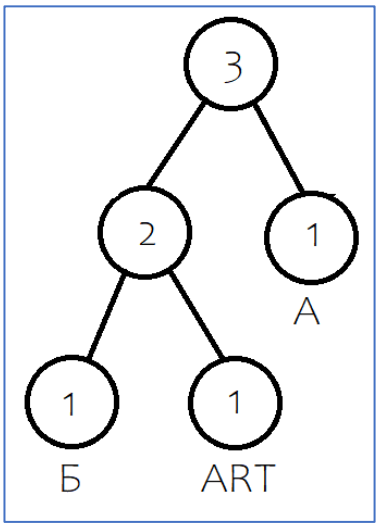
3) Дальше считываем «Б». Опять в дереве нет, отправляем код искусственного символа (1, так как идём вправо) и код Б. Бьём вершину. Битьё вершины подразумевает увеличение частоты предков на 1, а также добавление детей в массив.



Результат: code[A] **1** code[Б].

Вершина	ART_0	A	ART_1	Б	ART
idex	0	1	2	3	4
частота	2	1	2	1	1

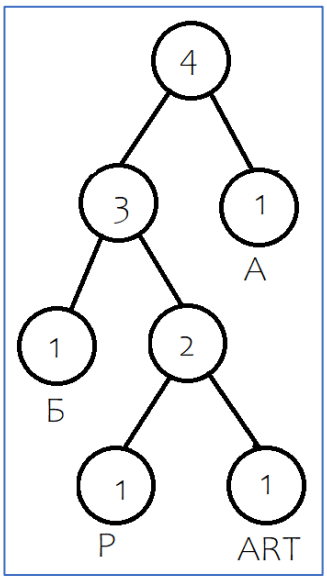
4) Так как для всех детей слева направо должен быть порядок не возрастания, меняем детей местами.



Результат: code[A] **1** code[Б].

Вершина	ART_0	ART_1	A	Б	ART
idex	0	1	2	3	4
частота	2	2	1	1	1

5) Добавляем «Р».



Результат: code[A] **1** code[Б] **01** code[Р].

6) Нарушено условие не возрастания.

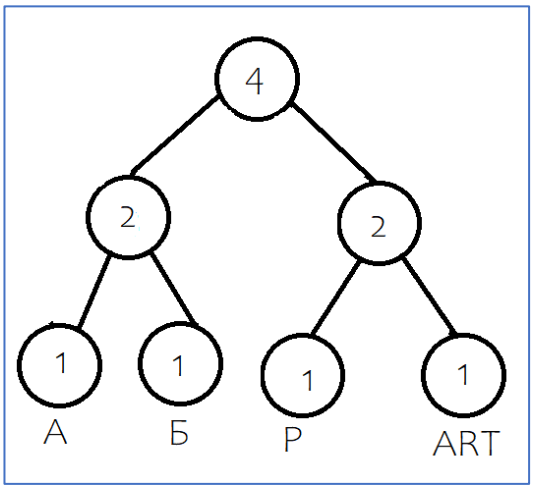
Вершина	ART_0	ART_1	A	Б	ART_2	Р	ART
idex	0	1	2	3	4	5	6
частота	3	2	1	1	2	1	1

Нарушено условие не возрастания.

Вершина	ART_0	ART_1	A	ART_2	Б	Р	ART
idex	0	1	2	3	4	5	6
частота	3	2	1	2	1	1	1

Теперь всё хорошо.

Вершина	ART_0	ART_1	ART_2	A	Б	Р	ART
idex	0	1	2	3	4	5	6
частота	4	2	2	1	1	1	1



Результат: code[А] **1** code[Б] **010** code[Р].

Результат: code[А] **1** code[Б] **01** code[Р] **01**.

7) Добавляем «А». Что мы для этого делаем? Да просто обновляем значения.

Вершина	ART_0	ART_1	ART_2	А	Б	Р
idex	0	1	2	3	4	5
частота	5	3	2	2	1	1

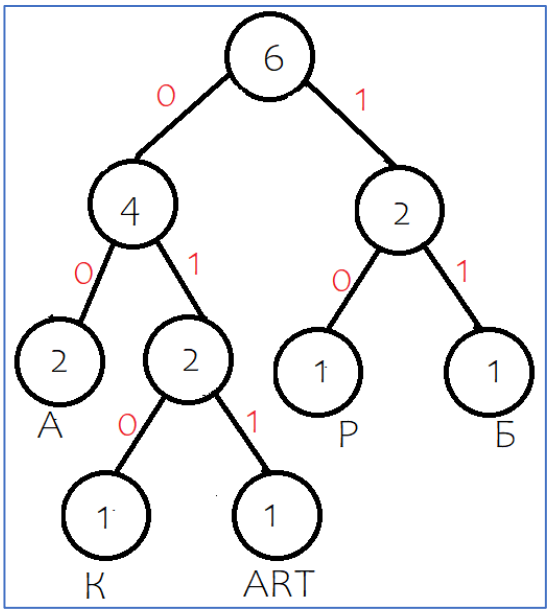
8) Добавляем «К».

Вершина	ART_0	ART_1	ART_2	А	Б	Р	ART_3	К	ART
idex	0	1	2	3	4	5	6	7	8
частота	4	2	2	2	1	1	2	1	1

По итогу имеем:

Вершина	ART_0	ART_1	ART_2	А	ART_3	Б	Р	К	ART
idex	0	1	2	3	4	5	6	7	8
частота	6	4	2	2	2	1	1	1	1

Результат: code[А] **1** code[Б] **01** code[Р] **01 11** code[К].



Вот и вся идея.

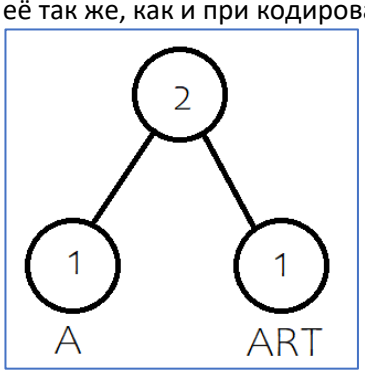
Декодирование:

Есть строка code[X_1] 1 code[X_2] 01 code[X_3] 0111 code[X_4].

Декодируем так же, как и кодируем.

Строим дерево. Сначала в нём есть только искусственный символ. Что мы делаем: пришли в существующий лист (достигли его) – в результат пишем эту букву, пришли в несуществующий – создаём букву, закодированную следующими 8-ю битами.

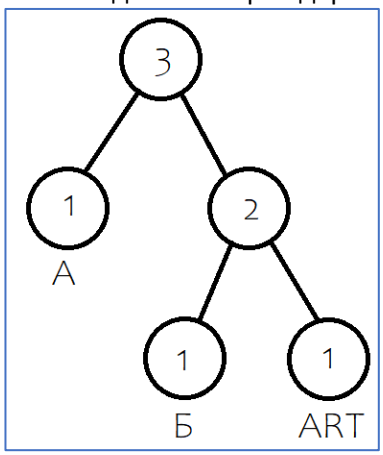
0) Мы сначала в искусственном символе. Считываем код буквы X_1 – это А. Теперь нужно добавить её так же, как и при кодировании.



Результат: А.

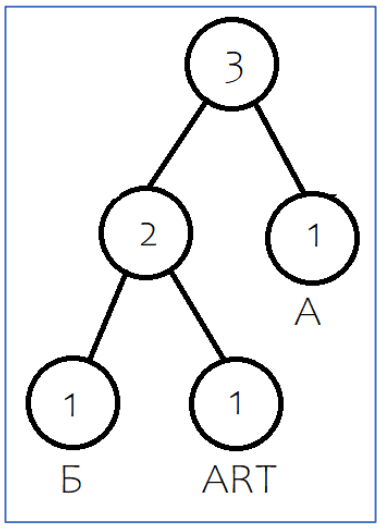
Вершина	ART_0	А	ART
idex	0	1	2
частота	2	1	1

1) Мы находимся в корне дерева. Это не лист. Считываем 1, идём направо. Считываем X_2 – это Б.



Результат Б.

Вершина	ART_0	А	ART_1	Б	ART
idex	0	1	2	3	4
частота	2	1	2	1	1



Результат: Б.

Вершина	ART_0	ART_1	А	Б	ART
idex	0	1	2	3	4
частота	2	2	1	1	1

2) То есть алгоритм точно такой же, что и при кодировании, только смотрим, куда идти. Небольшой нюанс: 01 11, мы пришли по 01 в А, записали в результат, вернулись в корень, а потом ищем 11.

Максимальная частота встречаемости в массиве – 1 – это количество букв в слове.

Адаптивный код Хаффмана для «КАРМА»

1	2	3	4	5	6	7	8	9
ART_0	ART_1	ART_2	ART_3	А	Р	К	М	ART
6	4	2	2	2	1	1	1	1

Адаптивный код Хаффмана для «АВВАВ»

1	2	3	4	5
ART_0	ART_1	В	А	ART
6	3	3	2	1

14. Канонические коды Хаффмана. Вычисление длин кодов.

Ключ: однозначное дерево, сортируем буквы лексикографически, считаем биты, потом кодируем. Дописываем ячейки в хип для хранения указателей, вынимаем два минимальных элемента, объединяем, делаем ссылку. Получим биты. Коды: сортируем убыванию числа бит, потом лексикографически. Число бит – код 1, если бит одинаково с кем-то, кто выше +1. Для декодирования хранить сколько бит для буквы.

Особенность: в полустатических кодах надо брать две вершины, с одинаковой частотой. А что, если их несколько? Берём наугад. А можно ли сделать так, чтобы выбор был однозначен? Канонические коды Хаффмана говорят, что можно. Это позволяет построить однозначное дерево.

Идея алгоритма кодирования:

1) Сортируем словарь букв в лексикографическом порядке. После этого считаем, сколько бит нужно, чтобы закодировать каждую буковку.
2) После того, как узнали, сколько бит нужно, определяем однозначно то, что надо сделать, чтобы получить код буквы.
3) То есть сначала биты, потом коды.

Пример:

Строка «АБРАКАДАБРА».

1) Сортируем буквы в лексикографическом порядке:
   АБДКР
2) Считаем, сколько раз каждая буква встречается.

А	Б	Д	К	Р
5	2	1	1	2

3) У нас есть пять букв. Слева, к нашему массиву, дописываем пять ячеек. В них мы храним указатели на буквы (позиции в массиве).

					А	Б	Д	К	Р	
5	6	7	8	9	5	2	1	1	2	
0	1	2	3	4	5	6	7	8	9	i

4) Мы строим хип по первой половине массива, но сортируем по тому, на что ссылаются элементы.

Пример:

1. Берём середину массива - 7. Дети стоят на позициях 2i+1, 2i+2. У 7 детей нет. Идём дальше.

5	6	7	8	9
0	1	2	3	4

А	Б	Д	К	Р	
5	2	1	1	2	
5	6	7	8	9	i

2. Берём 6. У него дети с индексами 3 и 4.

5	6	7	8	9
0	1	2	3	4

А	Б	Д	К	Р	
5	2	1	1	2	
5	6	7	8	9	i

Смотрим: сравниваем элемент массива с индексом 6 с индексами 8 и 9 (2 с 1 и 2). Выбираем минимум. Меняем местами элементы хипа так, чтобы минимум оказался на позиции 1. То есть 6 и 8 местами меняем. Больше ничего не делаем! Индексы массива с 5 по 9 не трогаем!

5	8	7	6	9
0	1	2	3	4

А	Б	Д	К	Р	
5	2	1	1	2	
5	6	7	8	9	i

3. И так далее. Получим:

7	8	5	6	9
0	1	2	3	4

А	Б	Д	К	Р	
5	2	1	1	2	
5	6	7	8	9	i

5) Мы знаем, что первый элемент хипа – самый маленький. Поменяем его с последним элементом хипа, и уменьшим размер хипа на 1.

9	8	5	6
0	1	2	3

	А	Б	Д	К	Р	
7	5	2	1	1	2	
4	5	6	7	8	9	i

6) Но тогда наш хип нарушается, это уже не хип. Приводим его к необходимому виду

8	9	5	6
0	1	2	3

	А	Б	Д	К	Р	
7	5	2	1	1	2	
4	5	6	7	8	9	i

7) Опять меняем элемент с наименьшей частотой с последним.

6	9	5
0	1	2

		А	Б	Д	К	Р	
8	7	5	2	1	1	2	
3	4	5	6	7	8	9	i

8) Теперь делаем следующее: из двух минимальных вынутых элементов хипа мы образуем новый (путём сложения частот букв, стоящих на позициях вынутых чисел в массиве), который помещаем к буквам, а в хип помещаем ссылку на этот элемент.
На примере: мы вынули элементы «7» и «8». Сложим частоты элементов на позициях 7 и 8 – 1+1=2. Поместим это значение на место элемента «8». В хип помещаем ссылку на ячейку (на i = 4).

6	9	5	4
0	1	2	3

Д+К	А	Б	Д	К	Р	
2	5	2	1	1	2	
4	5	6	7	8	9	i

У нас может быть хип – не хип. Проверяем: ничего не нарушено.
Далее мы меняем значения ячеек 7 и 8 на ссылку на ячейку с их общей частотой встречаемости:

6	9	5	4
0	1	2	3

Д+К	А	Б	Д	К	Р	
2	5	2	4	4	2	
4	5	6	7	8	9	i

9) Повторяем пункты 5-8 три раза.

Первый круг:

4	9	5
0	1	2

	Д+К	А	Б	Д	К	Р	
6	2	5	2	4	4	2	
3	4	5	6	7	8	9	i

Проверяем хип:

5	9
0	1

		Д+К	А	Б	Д	К	Р	
4	6	2	5	2	4	4	2	
2	3	4	5	6	7	8	9	i

5	9	3
0	1	2

Д+К+Б	Д+К	А	Б	Д	К	Р	
4	3	5	3	4	4	2	
3	4	5	6	7	8	9	i

Корректируем хип:

9	5	3
0	1	2

Д+К+Б	Д+К	А	Б	Д	К	Р	
4	3	5	3	4	4	2	
3	4	5	6	7	8	9	i

Второй круг:

5
0

		Д+К+Б	Д+К	А	Б	Д	К	Р	
3	9	4	3	5	3	4	4	2	
1	2	3	4	5	6	7	8	9	i

5	2
0	1

Д+К+Б+Р	Д+К+Б	Д+К	А	Б	Д	К	Р	
6	2	3	5	3	4	4	2	
2	3	4	5	6	7	8	9	i

Третий круг:

		Д+К+Б+Р	Д+К+Б	Д+К	А	Б	Д	К	Р	
2	5	6	2	3	5	3	4	4	2	
1	0	2	3	4	5	6	7	8	9	i

1
0

Д+К+Б+Р+А	Д+К+Б+Р	Д+К+Б	Д+К	А	Б	Д	К	Р	
11	1	2	3	1	3	4	4	2	
1	2	3	4	5	6	7	8	9	i

10) Теперь хип состоит из одного элемента, указывающий на корень следующего массива. В массиве мы получили числа. Что это за числа? С их помощью мы будет считать длину кода. Как?

Д+К+Б+Р+А	Д+К+Б+Р	Д+К+Б	Д+К	А	Б	Д	К	Р	
11	1	2	3	1	3	4	4	2	
1	2	3	4	5	6	7	8	9	i

Для буквы «А»: из пятого индекса идём в первый. Считаем, сколько шагов потребовалось, чтобы это сделать? Один. Биты для буквы А – 1.

Для «Б»: из 6 индекса идём в 3, из в 3 во 2, из 2 в 1. Три шага – биты «Б» = 3.

И так далее. Получим:

code_bit[A] = 1;

code_bit [Б] = 3;
code_bit [Д] = 4;
code_bit [К] = 4;
code_bit [Р] = 2.
Столькими битами должна кодироваться буква.

11) Альтернативный способ пункта 10.

Д+К+Б+Р+А	Д+К+Б+Р	Д+К+Б	Д+К	А	Б	Д	К	Р	
11	1	2	3	1	3	4	4	2	a
1	2	3	4	5	6	7	8	9	i

Вместо 11 ставим 0. Что делаем: идём слева направо по i, следуя алгоритму:
Смотрим, какое значение стоит на позиции ячейки a[i]. Переходим к j = a[i]. a[i] = b[j]+1.
То есть:
i=2

Д+К+Б+Р+А	Д+К+Б+Р	Д+К+Б	Д+К	А	Б	Д	К	Р	
0	1	2	3	1	3	4	4	2	a
1	2	3	4	5	6	7	8	9	i

i=3

Д+К+Б+Р+А	Д+К+Б+Р	Д+К+Б	Д+К	А	Б	Д	К	Р	
0	1	2	3	1	3	4	4	2	a
1	2	3	4	5	6	7	8	9	i

i=4

Д+К+Б+Р+А	Д+К+Б+Р	Д+К+Б	Д+К	А	Б	Д	К	Р	
0	1	2	3	1	3	4	4	2	a
1	2	3	4	5	6	7	8	9	i

i=5

Д+К+Б+Р+А	Д+К+Б+Р	Д+К+Б	Д+К	А	Б	Д	К	Р	
0	1	2	3	1	3	4	4	2	a
1	2	3	4	5	6	7	8	9	i

И так далее. Ответ будет:

Д+К+Б+Р+А	Д+К+Б+Р	Д+К+Б	Д+К	А	Б	Д	К	Р	
0	1	2	3	1	3	4	4	2	a
1	2	3	4	5	6	7	8	9	i

12) Как кодировать?
Дано:
code_bit[А] = 1;
code_bit [Б] = 3;
code_bit [Д] = 4;
code_bit [К] = 4;
code_bit [Р] = 2.

Сортируем все буквы: вначале по убыванию числа бит, внутри одинаковых бит – в лексикографическом порядке.

code_bit [Д] = 4; нужно 4 бита - 0000
code_bit [К] = 4; нужно 4 бита, но на 1 больше, чем предыдущее значение 0001
code_bit [Б] = 3; нужно 3 бита. Если есть буква с битом на 1 больше (4), то «откусываем» у него первую цифру - 001

code_bit [Р] = 2; нужно 2 бита. Если есть буква с битом на 1 больше (3), то «откусываем» у него первую цифру - 01

code_bit[А] = 1. нужен 1 бит. Если есть буква с битом на 1 больше (2), то «откусываем» у него первую цифру – 1.

code[А] = 1;
code[Б] = 001;
code[Д] = 0000;
code [К] = 0001;
code[Р] = 01.

Пример-ответ:

Для «КАРМА» кодировка:

К - 000

М - 001

Р – 01

А – 1

Идея алгоритма декодирования + пример:

Пусть есть алфавит, для которого мы получили количество бит:

A - 5
B - 3
C - 2
D - 5
E - 5
F - 3
G - 2
H - 5

Тогда кодировка будет:

A – 00000
D - 00001
E – 00010
H - 00011
B – 001
F - 010
C - 10
G - 11

Пусть мы закодировали AGFEH. Тогда получим 000001101000010000011.

Как декодировать? Можем точно так же посчитать коды, декодировать по ключу. Но это сложно.

Можно сделать так: сохранить буквы в таком порядке имеют длину кода такую-то. То есть,

A – 5, 0. D – 5,1. E – 5, 3. H – 5, 4. B – 3, 1. И так далее.

Пример:

1) **0**0000110100001000011.
   Есть ли буква, которая закодирована ровно одним битом? Нет. Смотрим дальше.
2) **00**0000110100001000011.
   Есть ли буква, которая закодирована двумя битами? Есть, но значение кода 00 < 2 (10) – поэтому это не C и не G. Смотрим дальше.
3) **000**0011010000100011.
   Есть ли буква, которая закодирована тремя битами? Есть. Но 000 < 1 (001).
4) **0000**0110100001000011.
   Есть ли буква, которая закодирована четырьмя битами? Нет.
5) **00000**1101000010000011.
   Есть ли буква, которая закодирована пятью битами? Есть. 00000 = 0, 0. Значит, это A.
6) И так далее.

15. Арифметическое кодирование. Статическая, полустатическая и адаптивная модели.

Ключ: бьём отрезок пропорционально вкладу, по буквам то же самое, храним границу и количество символов в полуинтервале. Декодируем аналогично.

Арифметическое кодирование

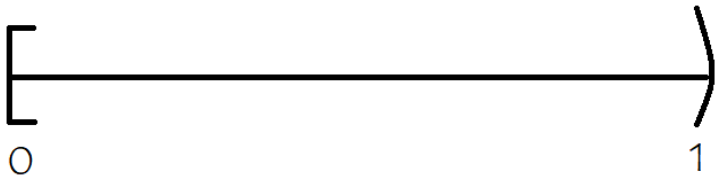
Идея:

Берём текст. Например, «КОВ. КОРОВА». Мы считаем, сколько раз каждая буква встречается.

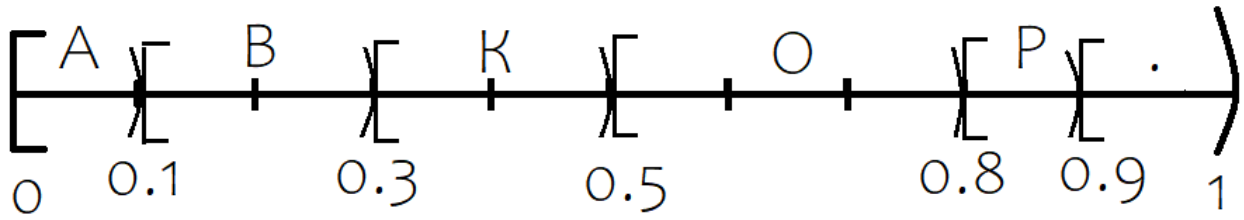
S = КОВ. КОРОВА

A	B	К	О	Р	.	
1	2	2	3	1	1	Total=10

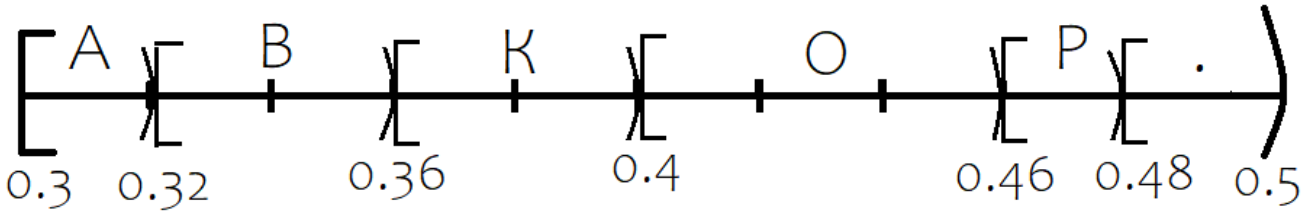
Далее берём интервал от 0 до 1. И бьём его пропорционально вкладу каждой буквы.



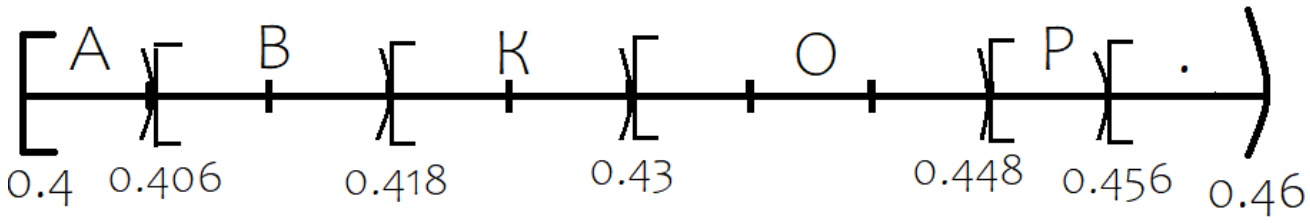
1. То есть:



2. После берём фразу (КОВ. КОРОВА), смотрим: первая буква «К». Определяем, какому полуинтервалу она принадлежит? От (0.3; 0.5]. «Вытаскиваем» его. Что делаем дальше? Разбиваем его так же на части по исходной строке.



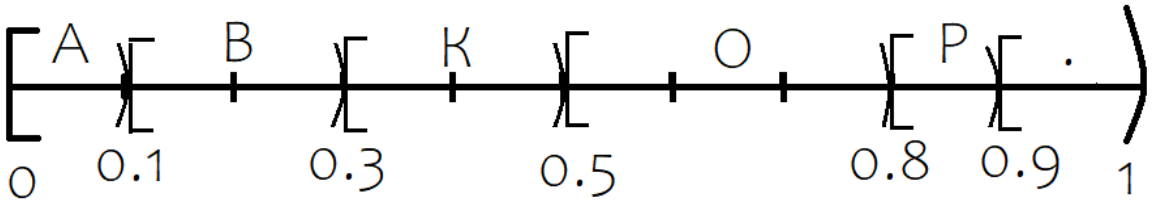
3. Следующая буква «О».



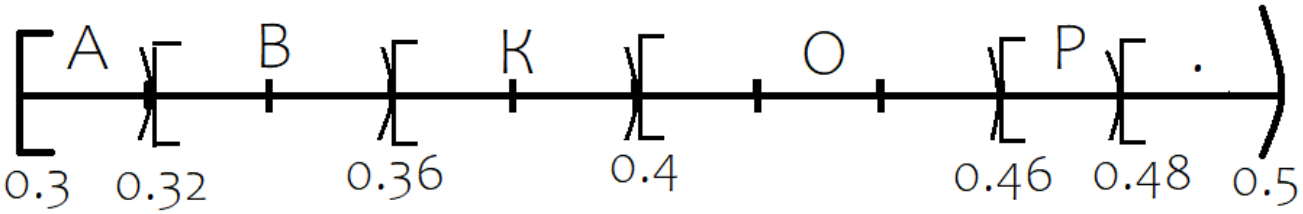
4. И так далее. Получим в конце интервал от некоего a до некого b.
   Способы хранения:
   1) Выбираем для кодировки пару (border, count), Border - число, принадлежащее [a; b) – можно просто взять a, count - количество символов за ним в полуинтервале.
      Если бы мы остановились на букве «В», то можно взять (0.406, 3)). Так как 0.406 принадлежит от [0; 1), после него идёт три символа (О, Р, .).
   2) Можно взять и поставить в конец строки сентинеллу, тогда левой границей можно передавать его.

Попробуем раскодировать пару (0.406, 3). У нас А – 1/10, В – 2/10, К – 2/10, О – 3/10, Р – 1/10, . – 1/10.

Что мы делаем: берём отрезок от 0 до 1 и делаем разбиение.



Смотрим: у нас число 0.406. Какому интервалу оно принадлежит? 0.3-0.5, буква «К». Это и будет первая буква. Дальше делаем как в кодировании – бьём на отрезки.



Какому интервалу принадлежит 0.406? Буква О. Получаем «КО». Теперь бьём «В». И всё бы хорошо, но у нас после буквы О останется три символа, а это предел нашего кодирования. Останавливаемся. Получили «КОВ». Не весь исходный пример, так как мы не всё закодировали.

Как можно оптимизировать?

Чем больше мы делим, тем хуже работает алгоритм. Поэтому можно брать отрезок от 0 до, допустим, 1023. Более-менее всё будет в целых числах.

Можно для левой и правой границы для отрезка «выкидывать биты». Например, есть отрезок от 253 до 255. В битах это **1111**00110000 и **1111**1111111 соответственно. Заметим, что первые биты повторяются. Оставим только 00110000 и 11111111.

Арифметическое кодирование может быть статическим и полустатическим. В статическом случае все буквы занимают одинаковые отрезки без учёта количества повторов. В полустатическом – считаем.

16. Преобразование Барроуза-Уилера.

Ключ: выпишем все циклические сдвиги строки лексикографически, выписываем последние буквы - столбец + позиция искомого слова - БУ. Сложность O(сортировка). Сортим лексикографически столбец, соединяем столбец + отсортированный столбец, сортим, опять соединяем с исходным последним столбцом и тп. Так восстановим исходные строки и по позиции найдём исходное слово. $O(n^3 \log(n))$.

Соответствие позиций отсортированных и неотсортированных. Смотрим букву на позиции со входа преобразования, пишем букву в ответ, смотрим позицию в i, ищем i = i_mod, пишем в ответ букву и тд. O(n+n) = (сортировка + соответствия) = O(n).

Можно ли производить предобработку текста так, чтобы он лучше сжимался?

Можно. Рассмотрим преобразование Барроуза-Уилера. Его суть в том, чтобы преобразовать слово так, чтобы стандартный алгоритм сжатия справился лучше, нежели с обычным словом.

Как произвести это преобразование?

//Длина кода «АБРАКАДАБРА» полустатическим Хаффманом - 23 бита.

Пусть «АБРАКАДАБРА» - циклическая строка. Тогда будем иметь такие сдвиги:

АБРАКАДАБРА
БРАКАДАБРАА
РАКАДАБРААБ
АКАДАБРААБР
КАДАБРААБРА
АДАБРААБРАК
ДАБРААБРАКА
АБРААБРАКАД
БРААБРАКАДА
РААБРАКАДАБ
ААБРАКАДАБР

Отсортируем в лексикографическом порядке:

0. ААБРАКАДАБР
1. АБРААБРАКАД
2. АБРАКАДАБРА
3. АДАБРААБРАК
4. АКАДАБРААБР
5. БРААБРАКАДА
6. БРАКАДАБРАА
7. ДАБРААБРАКА
8. КАДАБРААБРА
9. РААБРАКАДАБ
10. РАКАДАБРААБ

Теперь в каждой строке отсортированного списка берём последнюю букву и записываем все их в строку:

РДАКРАААААББ

Наше искомое слово «АБРАКАДАБРА» стоит на 2-й позиции.

Запишем (РДАКРАААААББ, 2) – это и есть преобразование Барроуза-Уилера.

Назревает вопрос: и что с этим делать?

Идея такая: допустим, есть многобуквенное слово в тексте. И встречается оно несколько раз. После того, как отсортируем сдвиги, мы как можно ближе ставим друг к другу одинаковые буквы (которые оказываются в конце из-за дробления слова).

Сложность алгоритма: сложность сортировки (например, поразрядная за $O(n^2)$) или строим суффиксный массив за $O(n)$.

1) Мы получили такую строку РДАКРАААААББ. Раз это последний столбец, то это все буквы, которые встречались в тексте. Отсортируем их в порядке возрастания:

   ААААААББДКРР.

2) И это первые буквы, так как ты отсортировали строки. Также мы знаем, что за последней буквой соответственно будет следовать первая. То есть:

   Р Д А К Р А А А А Б Б

   А А А А А Б Б Д К Р Р

   =>

   0. РА
   1. ДА
   2. АА
   3. КА
   4. РА
   5. АБ
   6. АБ
   7. АД
   8. АК
   9. БР
   10. БР

3) Отсортируем теперь эти строки.
   0. АА
   1. АБ
   2. АБ
   3. АД
   4. АК
   5. БР
   6. БР
   7. ДА
   8. КА
   9. РА
   10. РА
4) Эти строки опять соединяем с последним столбцом РДАКРАААААББ и опять сортируем (сначала буква последнего столбца, потом строка).
   0. ААБ
   1. АБР
   2. АБР
   3. АДА

4. АКА
5. БРА
6. БРА
7. ДАБ
8. КАД
9. РАА
10. РАК

5) Опять дописываем к последнему столбцу и т д. И так пока не получим строки длины n. Мы по итогу мы должны получить ровно те же строки, что и в начале:
   0. ААБРАКАДАБР
   1. АБРААБРАКАД
   2. АБРАКАДАБРА
   3. АДАБРААБРАК
   4. АКАДАБРААБР
   5. БРААБРАКАДА
   6. БРАКАДАБРАА
   7. ДАБРААБРАКА
   8. КАДАБРААБРА
   9. РААБРАКАДАБ
   10. РАКАДАБРААБ

   Таким образом мы можем восстановить наши строки.

Итоговая сложность – $O(n^3 \log(n))$. Можно ли быстрее? Можно.

Обратное преобразование Б-У.

Рассмотрим ещё раз к S = (Р Д А К Р А А А А Б Б, 2)

Отсортируем, и поставим в соответствие правому столбцу итерации первого (они не должны повторятся):

i	sorted	not_sorted	i_mod
0	А	Р	9
1	А	Д	7
2	А	А	0
3	А	К	8
4	А	Р	10
5	Б	А	1
6	Б	А	2
7	Д	А	3
8	К	А	4
9	Р	Б	5
10	Р	Б	6

Теперь смотрим на вторую позицию (т к 2 на входе было). На ней стоит А – с неё всё началось. Ищем вторую итерацию в i_mod. Напротив неё в первом столбце стоит Б на 6 позиции. Это будет буква после А, итого «АБ». Ищем 6 в i_mod. Напротив стоит Р на позиции 10. Итого «АБР». И так далее мы восстановим «АБРАКАДАБРА».

Общая сложность обратного преобразования Барроуза-Уильяма – O(n+n) = (сортировка + соответствия) = O(n).

Ради чего мы это всё делаем? Однозначность воспроизведения, уменьшение объёмов текста (кода?) и тп.

17. Преобразование run-length encoding.

Ключ: вместо подряд повторяющихся букв – буква + количество.

Есть алгоритм, альтернативный “Move To Front” – «RLE» (Run Length Encoding).

В чём идея: если подряд идут несколько одинаковых букв, то вместо того, чтобы кодировать эту букву несколько раз, то кодируем её буквой, а потом столько раз, сколько раз она встретилась. Исключение – когда встречается один раз.

Тогда S = Р Д А К Р А А А А Б Б кодировалось бы так: Р Д А К Р А 4 Б 2.

18. Преобразование Move to Front.

Ключ: сортим буквы в лексикографическом, два создаём массива(буквы и вспомогательный), идём по столбцу, смотрим, на какой позиции стоит буква из массива букв, сдвигаем её в начало, позицию, на которой стояла, пишем в вспомогательный массив. Считаем кол-во 0-4 – чем чаще встречается, тем меньше бит. Строим дерево Хаффмана. Декодируем так же.

Вот наш столбец: Р Д А К Р А А А А Б Б (вопрос 16).

Есть такая вещь, как “Move To Front”, применяем после преобразования Барроуза-Уилера.

Берём буквы в лексикографическом порядке:

А	Б	Д	К	Р	
0	1	2	3	4	i

Идём по столбцу и смотрим, на какой позиции i стоит наша буква. И вместо буквы пишем число. После чего двигаем букву в начало.

Пример:

1) S = Р Д А К Р А А А А Б Б.

	А	Б	Д	К	Р	
	0	1	2	3	4	i

S_modified = 4.

	Р	А	Б	Д	К	
	0	1	2	3	4	i

2) S = Р Д А К Р А А А А Б Б.

	Р	А	Б	Д	К	
	0	1	2	3	4	i

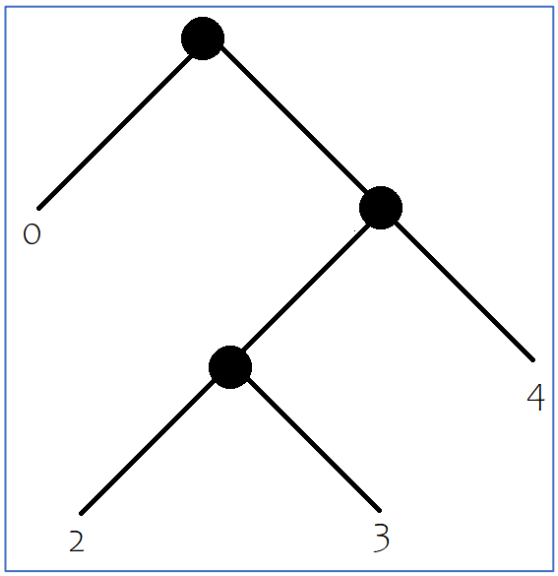
S_modified = 4 3.

	Р	Д	А	Б	К	
	0	1	2	3	4	i

3) И так далее. Получим:
   S_modified = 4 3 2 4 3 2 0 0 0 4 0.
4) Считаем количество нулей двоек, троек, четвёрок.

«0» – 4
«2» - 2
«3» - 2
«4» - 3

5) Строим дерево Хаффмана:



Итого сколькими битами что будет кодироваться:
«0» – 1 бит
«2» - 3 бита
«3» - 3 бита
«4» - 2 бита
Всего 4*1+2*3+2*3+3*2 = 22 бита. То есть работает лучше.

Как декодировать? Да так же, как и кодировали.

S_modified = 4 3 2 4 3 2 0 0 0 4 0.
Набор наших букв:

	А	Б	Д	К	Р	
	0	1	2	3	4	i

1) Смотрим: первая 4. Это «Р», выписываем её, «Р» сдвигаем в начало.

	Р	А	Б	Д	К	
	0	1	2	3	4	i

2) И так далее.

19. Словарные методы сжатия. Семейство алгоритмов LZ-77.

Ключ: буфер, скользящее окно, тройка (двинуться на n позиций, взять с получившейся позиции k букв, добавить букву “letter”). Декодирование: тупо обратный процесс с буфера. Много повторяющихся букв – скользим и по коду.

Это словарные методы сжатия текста. Есть некоторый словарь. Мы берём слово, ищем его в словаре, условно подставляем вместо него его код. Декодирование аналогично. Обновляем параллельно словарик, то есть это адаптивный алгоритм.

Есть два семейства алгоритмов: LZ-77 (LZ-1), LZ-78(LZ-2). 77, 78 – года. Вклад больше Зив внёс.

Сначала смотрим **LZ-77.**

Идея: есть некоторый текст, одна часть его закодирована, а вторую надо закодировать.

Что мы делаем? Берём какое-то слово A в части, которую надо раскодировать, ищем его код в уже раскодированной части и подставляем код A из раскодированной части. И так далее.

Это алгоритм сжатия со скользящим окном. То есть, есть часть Encoded - «буфер» и правая часть, которую надо закодировать - меняется. Чем чаще повторяются слова, чем лучше.

Кодируем мы тройку: размер сдвига (чтобы сдвинуться в начало строки A), сколько символов из окна, текущий символ.

Пример, кодирование:

S =	a	b	a	a	b	a	b	a	b	b	a

0) Пока буфер 0.

a	b	a	a	b	a	b	a	b	b	a

После получаем тройку (0, 0, a) – сдвинулись на ноль символов, закодировали ноль символом и взяли букву a. То есть мы просто закодировали букву «a». Буфер содержит теперь «a».
Буфер:
(0, 0, a)

a	b	a	a	b	a	b	a	b	b	a

1) Дальше. Кодируем «b».

a	b	a	a	b	a	b	a	b	b	a

Есть «b» в буфере? Нет. Тогда имеем (0, 0, b) - сдвинулись на ноль символов, закодировали ноль букв и взяли букву b.
Буфер:
(0, 0, a)
(0, 0, b)

a	b	a	a	b	a	b	a	b	b	a

2) Кодируем дальше. Буква «a» есть? Да. Не делаем ничего. А «aa»? Нет. Что мы делаем: чтобы получить aa, надо сдвинуться влево на 2 от нашей позиции, от неё взять один символ, чтобы получить «a». И к ней надо добавить ещё «a».

a	b	a	a	b	a	b	a	b	b	a

Получим (2, 1, a).

a	b	a	a	b	a	b	a	b	b	a

Буфер:
(0, 0, a) = a
(0, 0, b) = b
(2, 1, a) = aa

3) Кодируем дальше. Буква «b» есть? Да. Слово «ba» есть? Да. Слово «bab» есть? Нет.

a	b	a	a	b	a	b	a	b	b	a

Кодируем: (3, 2, b) = bab.

a	b	a	a	b	a	b	a	b	b	a

Буфер:
(0, 0, a) = a
(0, 0, b) = b
(2, 1, a) = aa
(3, 2, b) = bab

4) И так далее. Как кодировать последний символ? Пустой строкой.
Имеем буфер:
(0, 0, a) = a
(0, 0, b) = b
(2, 1, a) = aa
(3, 2, b) = bab
(2, 2, b) = abb
(3, 1, ‘ ‘) = a

Пример, декодирование:

Имеем буфер:
(0, 0, a) = a
(0, 0, b) = b
(2, 1, a) = aa
(3, 2, b) = bab
(2, 2, b) = abb
(3, 1, ‘ ‘) = a

Поступаем обратным способом:

0) (0, 0, a) = a. Пишем a.

a

1) (0, 0, b) = b. Так как по буферу двигаться не надо, пишем b.

a	b

2) (2, 1, a) = aa. Мы должны сдвинуться по буферу на 2, прочитать один символ – a. Написать его, после приписать ещё a.

a	b	a	a

3) И так далее.

Пример 2 (тоже LZ-77, но чуть по-другому)

При кодировании мы скользим не только по окну, но и по закодированной части. Можно бежать и по буферу, и по коду.

Пусть есть строка: abbbbbbbbbbba.

Буфер:
(0, 0, a) = a
(0, 0, b) = b
(1, 1, b) = bb
(2, 2, b) = bbb
…
(1, 10, a) = a.

Мы можем записать буфер так:

(0, 0, a) = a
(0, 0, b) = b
(1, 10, a) = a.

Почему? Попробуем декодировать.

0) (0, 0, a) = a.

a

1) (0, 0, b) = b.

a	b

2) (1, 10, a) = a. А дальше мы должны написать 10 символов в предыдущем раскодировании. Буфер на чтение идёт на опережение. Итератор на чтение на b, а на запись – на следующей ячейке.

a	b	b	b	b	b	b	b	b	b	b	b

3) Ну и a в конец пишем.

Пример 3 (тоже LZ-77, но тоже с повторами)

Строка:

a	b	a	b	a	b	b	a	b	b	a

Буфер:

(0, 0, a)
(0, 0, b)
(2, 4, b)
(4, 4, a)

20. Словарные методы сжатия. Семейство алгоритмов LZ-78.

Ключ: если слово есть в словаре, приписываем букву, если есть – ещё приписываем, нет – кодируем слово без буквы, в словарь добавляем предыдущее добавленное в словарь слово с буквой. Пара – (к какому слову, какую букву дописали). Декодируем аналогично.

Идея:

В словаре всегда есть пустое слово, его код – 0. Мы всегда стараемся добавлять в словарь слово как можно большей длины: если слово есть в словаре, приписываем к нему ещё одну букву и смотрим, есть ли оно у нас в словаре. Если его нет, кодируем последнее слово, которое у нас было, ещё одну букву, которая следует за ним, а потом конкатенируем слово с этой буквой и его добавляем в словарь со следующим кодом. Мы кодируем пару.

Пример:

0	1	2	3	4	5	6	7	8	9
a	b	a	b	a	b	b	b	a	a

0) i = 0
Результат кодирования: (0, a) = a
Словарь: ø : 0
[ø + a = a] : 1

1) i = 1

Результат кодирования: (0, a) = a, (0, b) = b
Словарь: ø : 0
a : 1
[ø + b = b] : 2

2) i = 2
Результат кодирования: (0, a) = a, (0, b) = b, (1, b) = ab.
Словарь: ø : 0
a : 1
b : 2
[a + b = ab] : 3

3) i = 3
Результат кодирования: (0, a) = a, (0, b) = b, (1, b) = ab, (3, b) = abb.
Словарь: ø : 0
a : 1
b : 2
ab : 3
[ab + b = abb] : 4

4) i = 4
Результат кодирования: (0, a) = a, (0, b) = b, (1, b) = ab, (3, b) = abb, (2, a) = ba.
Словарь: ø : 0
a : 1
b : 2
ab : 3
abb : 4
[b + a = ba] : 5

5) i = 5
Результат кодирования: (0, a) = a, (0, b) = b, (1, b) = ab, (3, b) = abb, (2, a) = ba, (1, EOF) = a.
Словарь: ø : 0
a : 1
b : 2
ab : 3
abb : 4
ba: 5

Декодируем:

Вход: (0, a) = a, (0, b) = b, (1, b) = ab, (3, b) = abb, (2, a) = ba, (1, EOF) = a.
Словарь: ø : 0

Восстановим словарь по ходу декодирования.

1) (0, a) = a. (пустое множество и a – это a).
Словарь: ø : 0
a : 1
Результат: a

2) (0, b) = b. (пустое множество и b – это b).
Словарь: ø : 0
a : 1
b : 2
Результат: ab

3) (1, b) = ab. (буква a и b – это ab).
Словарь: ø : 0
a : 1

b : 2
ab : 3

Результат: abab

4) И так далее.

21. Алгоритм LZW. Способ эффективного расчета длины кодируемого слова.

Ключ: инициализация. Кодирование: когда слова + символы нет в словаре, кодирование + существующее слово, потом добавляем слово + символ в словарь. Декодирование: слово под кодом пишем в ответ, в словарь предыдущее декодированное + первая буква текущего декодированного.

Идея:

Рассмотрим сначала кодирование. У нас есть входная строка. С самого начала инициализируем словарь буквами, которые есть в ней. Далее мы обрабатываем поочерёдно каждый её символ следующим образом. Пусть мы рассматриваем символ на позиции i. Смотрим: есть ли в словаре слово, состоящее последовательно из символов на позициях от i до i+1? Допустим, есть. Тогда анализируем дальше: есть ли в словаре слово, состоящее из символов на позициях от i до i+2? Предположим, снова есть. Мы делаем такую проверку до тех пор, пока не встретим слово, не принадлежащее словарю. Пусть слово на позициях от i до i+n принадлежит словарю, а от i до i+n+1 - нет. Тогда мы производим два действия:

1. В результат кодирования добавляем код, соответствующий слову на позициях от i до i+n.
2. В словарь добавляем слово, состоящее из символов на позициях от i до i+n+1.

После этого обрабатываем символ на позиции i+1 аналогичным образом. Далее про декодирование. У нас есть входная строка чисел - закодированный текст.

С самого начала мы инициализируем словарь буквами алфавита. Далее мы обрабатываем поочерёдно каждое число таким образом. Пусть рассматриваемое число стоит на позиции i. Тогда мы делаем следующее:

1. В результат декодирования добавляем слово, соответствующее числу - коду.
2. В словарь добавляем слово, состоящее из предыдущего декодированного слова (код во входной строке - на позиции i-1) плюс первая буква текущего декодированного слова (его код во входной строке - на позиции i).

У алгоритма декодирования есть одна особенность: он как бы «отстаёт» от кодирования на один шаг. Это означает, что при таком алгоритме у нас может не быть слова, код которого соответствует последнему числу. Что делать в этой ситуации: исходя из алгоритма кодирования можем заметить, что предпоследнее закодированное слово (запишем его как a+X, где a - символ, X - последовательность символов) отличается от последнего закодированного на символ a. То есть последнее слово будет иметь вид: a+X+a.

Пример:

0	1	2	3	4	5	6	i
a	b	a	b	a	b	a	char

-1) Инициализируем словарь.

Словарь:

a : 1
b : 2

0) i = 0

Результат кодирования: 1

Словарь:

a : 1
b : 2

1) i = 1
   Результат кодирования: 1, 2
   Словарь:
   a : 1
   b : 2
2) i = 2
   Результат кодирования: 1, 2, 3.
   Словарь:
   a : 1
   b : 2
   ab : 3
3) i = 3
   Результат кодирования: 1, 2, 3, 3.
   Словарь:
   a : 1
   b : 2
   ab : 3
   ba : 4
4) i = 4
   Результат кодирования: 1, 2, 3, 5.
   Словарь: ø : 0
   a : 1
   b : 2
   ab : 3
   ba : 4
   aba : 5

Декодируем:

Инициализируем словарь: a : 1, b : 2.

Входная строка:

1	2	3	5	code
0	1	2	3	i

0) Результат кодирования: a
1) Результат кодирования: ab
   Словарь:
   a : 1
   b : 2
   ab : 3
2) Результат кодирования: abab
   Словарь:
   a : 1
   b : 2
   ab : 3
   ba : 4
3) Оп, а у нас нет числа 5 в словаре. Действуем согласно алгоритму: [ab+a = aba] : 5
   Результат кодирования: abababa
   Словарь:

a : 1
b : 2
ab : 3
ba : 4
aba: 5

Задача: закодировать LZW с помощью бинарного кода.

В бинарном виде мы делаем следующим образом: мы кодируем слово количеством битов, которыми можем закодировать длину текущего словаря.

То есть, для словаря:

a : 1
b : 2

Мы закодируем «a» как 001.

А для словаря:

ø : 0
a : 1
b : 2
ab : 3
ba : 4
aba: 5
abab: 6
bb: 7

«a» будет кодироваться как 0001, так как для кодирования 7+1 символов надо 4 бита.

Для строки «ACBCCAB» будет закодировано так:

001 011 010 011 011 0001 0010

При словаре:

ø : 0
A : 1
B : 2
C : 3
AC : 4
CB : 5
BC : 6
CC : 7

Раскодируем с учётом этой кодировки, учитывая количество символов в словаре плюс 1 (так как мы запаздываем).

Блок III. Графы

22. Графы, основные определения, способы представления

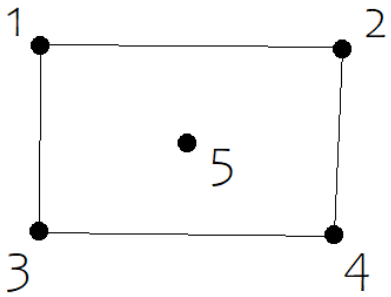
Граф – это пара из двух множеств v, e, G = (v, e), где v - вершины, e – рёбра, e = (v_i, v_j).

Его можно хранить с помощью:

1. Список смежностей – список для каждой вершины, где хранятся номера вершин, с которыми эта вершина инцидентна (расстояние – одно ребро).

   Список смежностей для графа:

   1: [2, 3];
   2: [1, 4];
   3: [1, 4];
   4: [2, 3];
   5: ø;

   В списках смежностей – 2*e элементов.

2. Матрица смежностей, размер v*v.

0	1	1	0	0
1	0	0	1	0
1	0	0	1	0
0	1	1	0	0
0	0	0	0	0

3. Сравнение способов хранения: список смежностей меньше по памяти, но в матрице искать путь проще.

Виды графов:

1. *Ориентированный граф* – граф, ребро которого имеет направление (e = <v_i, v_j>).
2. *Связный граф* – граф, в котором любая вершина достижима из любой другой.
3. *Взвешенный граф* – граф, ребро которого имеет стоимость. Например, за то, чтобы проехать по дороге А, надо заплатить 100 рублей, Б – 200 рублей и так далее.

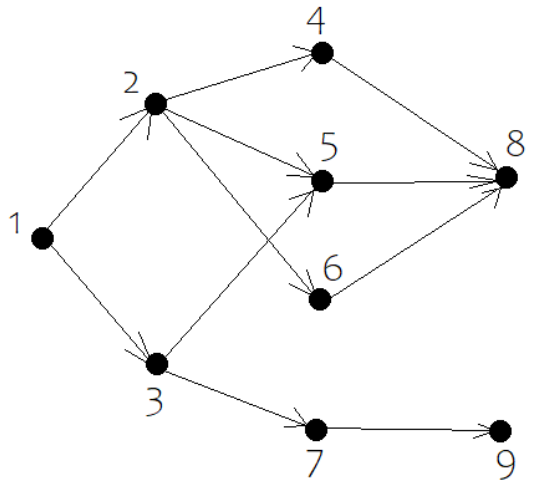
23. Поиск в ширину. Вычисление кратчайшего расстояния от одной вершины до остальных

Ключ: фронт волны, очередь текущей обработки Q и список посещённых вершин used. Проходим, за исключением used. O(v + e).

Поиск в ширину (BFS):

Поиск – как фронт волны. Мы начинаем из какой-то вершины и обходим все вершины графа постепенно. Берём вершину. Смотрим на все вершины, в которые мы можем перейти из текущей. Потом для первой посещённой вершины мы смотрим на все вершины, которые можем посетить из неё, исключая уже посещённые, потом то же для второй и так далее.

Пример:



Для обхода в ширину заводим очередь текущей обработки Q и список посещённых вершин used.

Запустим BFS(G, 1).

1) Q: [**1**], used = [1].
2) Q: [**2**, 3], used = [1, 2, 3].
3) Q: [**3**, 4, 5, 6], used = [1, 2, 3, 4, 5, 6].
4) Q: [**4**, 5, 6, 7], used = [1, 2, 3, 4, 5, 6, 7].
5) Q: [**5**, 6, 7, 8], used = [1, 2, 3, 4, 5, 6, 7, 8].
6) Q: [**6**, 7, 8], used = [1, 2, 3, 4, 5, 6, 7, 8].
7) Q: [**7**, 8], used = [1, 2, 3, 4, 5, 6, 7, 8].
8) Q: [**8**, 9], used = [1, 2, 3, 4, 5, 6, 7, 8, 9].
9) Q: [**9**], used = [1, 2, 3, 4, 5, 6, 7, 8, 9].
10) Q: [], used = [1, 2, 3, 4, 5, 6, 7, 8, 9].

Сложность O(v + e), v – очередь, e – один раз ребро рассматриваем.

Практическое применение: кратчайший путь.

Пусть в очередь мы будем класть пару: номер вершины и счётчик. Добавление смежных вершин +1.

1) Q: [**(1,0)**], used = [1].
2) Q: [**(2, 1)**, (3,1)], used = [1, 2, 3].
3) Q: [**(3, 1)**, (4, 2), (5, 2), (6, 2)], used = [1, 2, 3, 4, 5, 6].
4) Q: [**(4, 2)**, (5, 2), (6, 2), (7, 2)], used = [1, 2, 3, 4, 5, 6, 7].
5) Q: [**(5, 2)**, (6, 2), (7, 2), (8, 3)], used = [1, 2, 3, 4, 5, 6, 7, 8].
6) И так далее

Счётчик – за сколько можно дойти до вершины.

24. Поиск в глубину. Приложения поиска в глубину: топологическая сортировка, поиск сильно связных компонент.

Ключ: рекурсивно запускаем от непосещённого ребёнка, пока не закончились непосещённые или просто дети, тогда делаем шаг назад и опять рекурсия.

Поиск в глубину (DFS):

Запустим DFS(G, 1).
Рекурсивно запускаем DFS от не посещённого ребёнка: DFS(G, 2). Потом от его не посещённого ребёнка DFS(G, 4). Потом от его не посещённого ребёнка DFS(G, 8).

Рекурсия закончилась. Делаем шаг назад. Из 4-й вершины ничего не посещённого нет. Делаем шаг назад – вторая вершина. Есть не посещённые дети. Рекурсивно запускаем DFS от не посещённого ребёнка… И так далее.

НЕТ ПРИЛОЖЕНИЙ!!!

25. Кратчайшие пути из одной вершины графа. Влияние циклов на значение кратчайшего пути.

Путь в графе:

1. *Путь в графе* – набор пар рёбер, когда в каждом следующем ребре первая вершина является второй вершиной в предыдущем. (V1, V2), (V2, V3), (V3, V4)… То есть V1 -> V2 -> V3 -> V4… В пути у нас n вершин, n-1 рёбер.
2. *Простой путь* - путь без циклов.
3. *Цикл* – путь, в котором две вершины одинаковые, например, V1 -> V2 -> V1 -> V4…
4. *Минимальный путь* –
   Пусть стоимость ребра между вершинами V_{ij}, $V_{i(j+1)}$ записывается так: $W_{ij,i(j+1)}$. Тогда стоимость пути будет $W = \sum_{j=1}^{n-1} Wi_{j,}i_{j+1}$, минимальный путь – минимально возможная W.
5. Пути с циклами - **не оптимальные**, потому **что цикл будет иметь дополнительный вес**.
6. Если стоимость **цикла отрицательная**, то минимальный путь – **минус бесконечность**.

Поиск кратчайшего пути решается множеством вариантов.

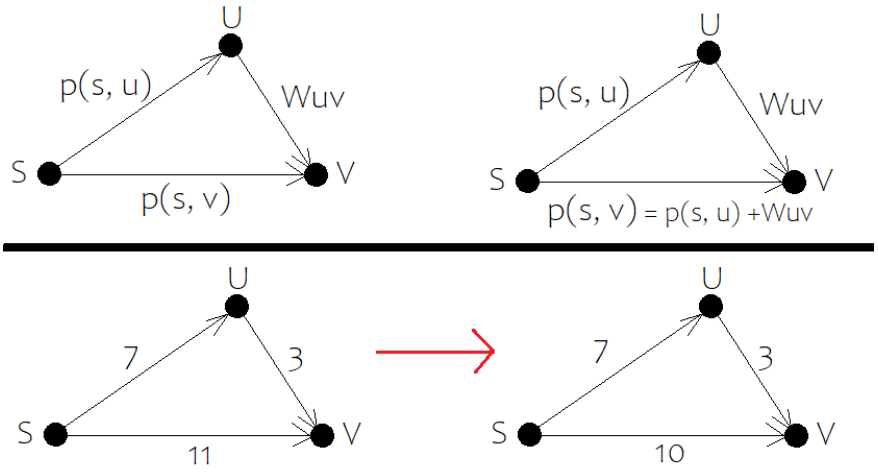
26. Алгоритм Беллмана-Форда.

Ключ: путь начальной 0, остальные ∞. Релаксируем, пока можем, иначе ответ. V-1 раз, иначе отриц цикл. O(E * (V – 1) + E) = O(V * E), релаксируем E рёбер V-1 раз. E в худшем случае примерно равна V^2.

Выкладка: что такое релаксация?

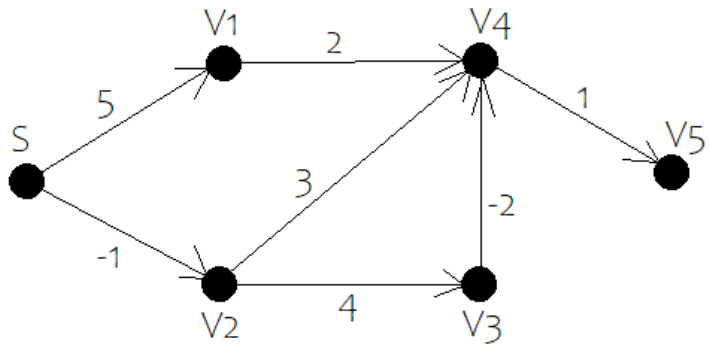
Если мы знаем, что для какого-то ребра знаем, что путь к началу этого ребра плюс переход по самому ребру меньше, чем путь из исходной вершины к концу этого ребра, мы заменяем путь к концу нашего ребра на путь к началу плюс наше ребро.

Если $P(S, V) > P(S, U) + W_{UV}$, то $P(S, V) = P(S, U) + W_{UV}$,

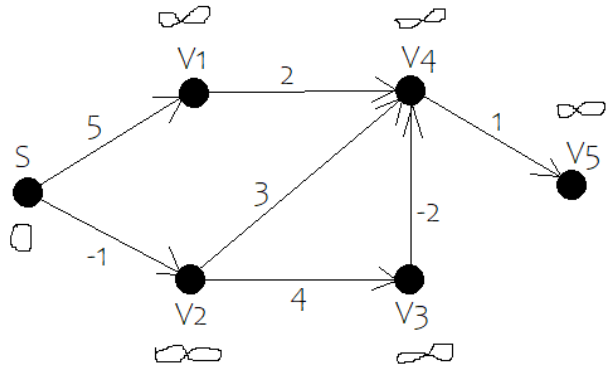


Алгоритм Белмана-Форда.

Есть граф:

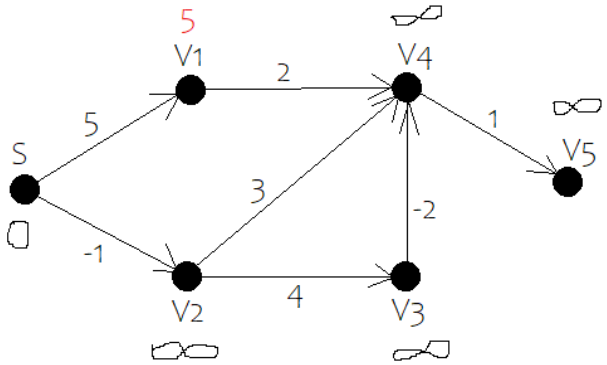


Пусть все вершины недостижимы (то есть путь бесконечность), а путь в S – 0. Запускаем релаксацию для всех рёбер.

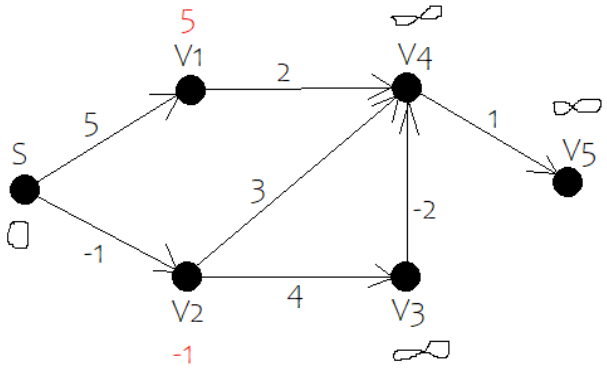


Первый цикл:

1) V_4, V_5 – кратчайший путь из S в V_4 бесконечность, из S в V_5 тоже. Бесконечность + 1 = бесконечность. Сокращать нечего.
2) (V_3, V_4), (V_2, V_3), (V_2, V_4), (V_1, V_4) – то же самое.
3) Из S в V_1 есть кратчайший путь из S в S – 0, прибавим 5, 0 + 5 = 5, из S в V_1 кратчайший путь бесконечность. 5 < бесконечности, поэтому минимальный вес = 5.

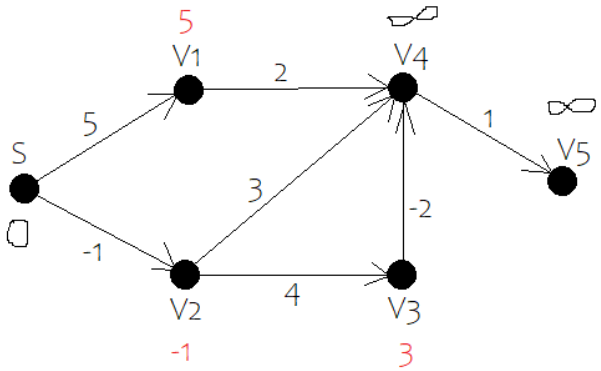


4) Из S в V_2 аналогично смотрим: 0 – 1 < ∞. Минимальный вес = -1.

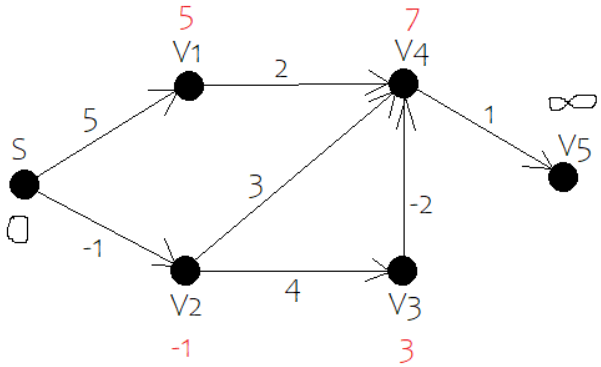


Второй цикл:

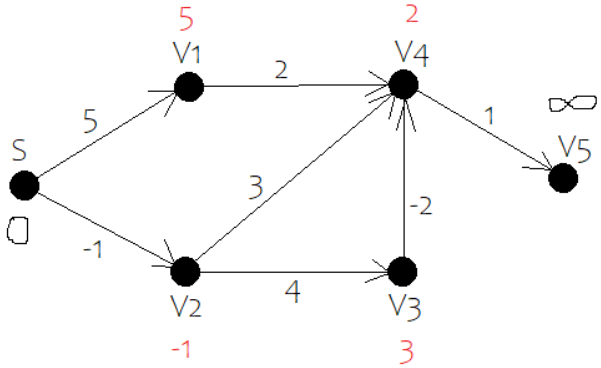
1) V_4, V_5 – кратчайший путь из S в V_4 бесконечность, из S в V_5 тоже. Бесконечность + 1 = бесконечность. Сокращать нечего. И так далее проверяем рёбра (примерно как в первом цикле, пока не дойдём до нужного)
2) Дошли до V_2, V_3. Из S в V_2 путь = -1. В V_3 бесконечность, -1 + 4 < ∞, пишем.



3) Дошли до V_1, V_4. 5+2 < ∞.

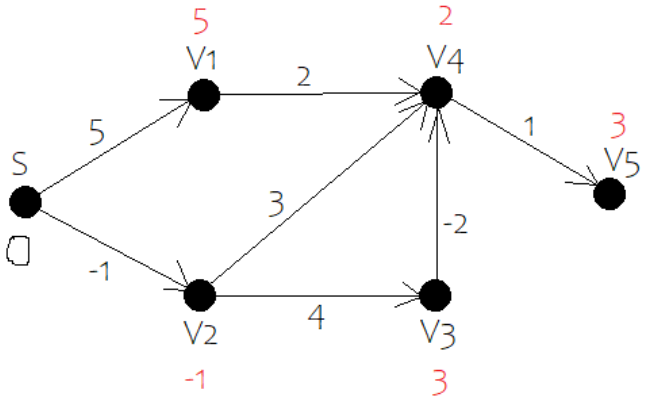


4) Так же не релаксировали ещё V_2, V_4. -1 + 3 < 7.

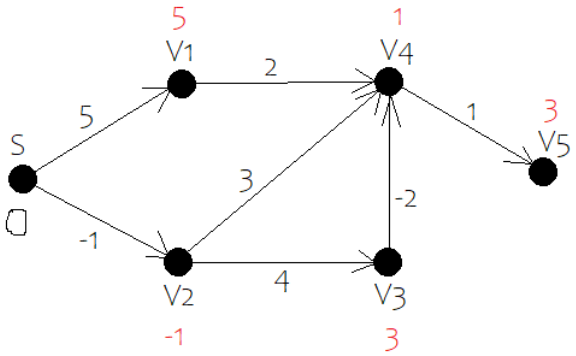


Цикл третий:

1) S-V_1, S-V_2, V_1-V_4, V_2-V_4, V_2-V_3 – не можем релаксировать.
2) V_4-V_5 мы можем релаксировать. $2+1 < \infty$.

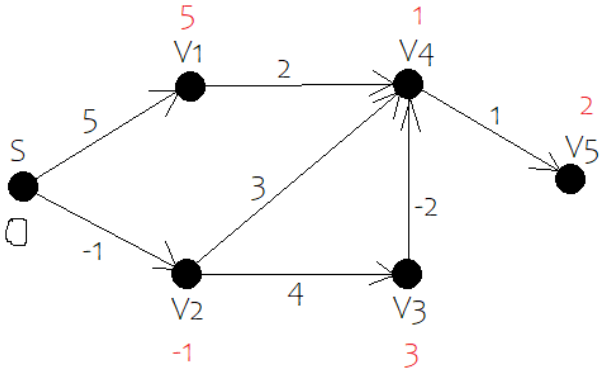


3) V_3-V_4 мы можем релаксировать. $3-2 < 2$.



Цикл пятый.

Идём по рёбрам, видим, что можем релаксировать V_4-V_5. $1+1<3$.



Цикл шестой.

Ни одно ребро не получается релаксировать, поэтому всё, мы получили кратчайшие пути.

Почему это работает?

Сначала все вершины были недостижимы, то есть пути – бесконечность. Когда мы первый раз релаксировали, у нас был путь из S в какую-то вершину, потом ещё. То есть мы так найдём кратчайшие пути длины один. На втором шаге у нас будут кратчайшие пути длиной не более 2. На

третьем – не более трёх. Если кратчайший путь не содержит циклов, то кратчайший путь будет содержать V-1 ребро. Поэтому V-1 раз просто применяется релаксация.

//В нашей реализации мы чуть считерили, в алгоритме Белмана-Форда мы берём элементы только из старого графа.

Если после V-1-й итерации вы запустили и получили какой-то новый кратчайший путь, то это значит, что существует отрицательный цикл.

Псевдокод:

```
For i = 0 to V-1
        For e in E
                Relax(s, e.first, e.second)
G' = G
For e in E
        Relax(s, e.first, e.second)
        Return G' = G.
```

То есть прогнали релаксацию V-1 раз, получили граф, сохранили его копию. Ещё один раз прогнали. Если получили несовпадение копии графа и графа (обновилось какое-то значение), то есть отрицательный цикл.

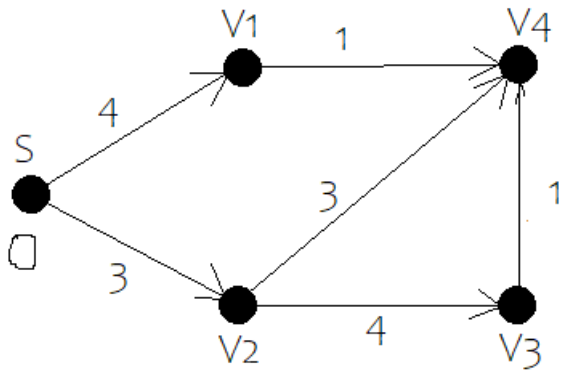
Сложность:

O(E * (V – 1) + E) = O(V * E), релаксируем E рёбер V-1 раз.

При этом E в худшем случае примерно равна V^2. Значит, сложность в худшем случае O(V * V^2) = O(V^3).

27. Алгоритм Дейкстры.

Ключ: ЖА, не для отрицательных рёбер, строим очередь с приоритетом, путь начальной 0, остальные ∞. Удаляем вершину с минимальным весов, релаксируем все рёбра из неё и так далее. O(V2 + E). На куче O(VlogV + ElogV). Если неполный граф, проще массив.



1) Мы строим из весов очередь с приоритетами или кучу.

S	V_1	V_2	V_3	V_4
0	∞	∞	∞	∞

2) Вытаскиваем S. Для всех рёбер, выходящих из вершины, запускаем релаксацию.

Релаксируем S-V_1, S-V_2, получаем соответственно 4 и 3.

V_2	V_1	V_3	V_4
3	4	∞	∞

3) Далее проделываем аналогичное действие с V_2.

V_1	V_3	V_4
4	6	7

4) Далее рассматриваем первую вершину.

V_3	V_4
5	7

5) Далее рассматриваем третью вершину.

V_4
6

6) И четвёртую. Элементы в очереди закончились, поэтому мы останавливаемся. Также мы могли бы остановиться в случае, если вес элемента – бесконечность.

Это жадный алгоритм, потому что мы всегда на каждом шаге выбираем минимальный вес пути.

Но есть проблема: алгоритм Дейкстры подразумевает отсутствие отрицательных рёбер (отрицательное ребро мы никак не учтём).

Сложность

На первом шаге мы проверяем V вершин, на втором шаге V-1, … на последнем – 1. Данная сумма пропорциональна V^2. То есть $O(V^2 + E)$. Обход рёбер и проверка вершин.

При этом E в худшем случае примерно равна V^2, так как $\frac{V(V+1)}{2}$. Значит, сложность в худшем случае $O(V^2 + V^2) = O(V^2)$. Это уже быстрее, чем Белман-Форд.

В случае использовании кучи, алгоритм Дейкстры будет работать за $O(V\log V + E\log V) = O(V^2\log V)$, так как обновляется ребро за логарифм и вытаскиваемся минимум за логарифм, то же с достать и вставить. Это быстрее, так как в очереди мы вытаскиваем минимум за V.

Если граф неполный, то проще использовать массив. Иначе на куче будет гораздо быстрее работать.

28. Кратчайшие пути между всеми парами вершин. Наивное решение через задачу о поиске кратчайших путей из одной вершины.

Алгоритм Белмана-Форда для такой задачи будет работать за $O(E * V)*V = O(E * V^2) \sim O(V^4)$.

Алгоритм Дейкстры: $O(E + V^2) * V \sim O(V^3)$.

Как можем быстрее или хотя бы так же? Есть несколько алгоритмов, например, алгоритм Флойд-Воршла и алгоритм Джонсона.

29. Задача о кратчайших путях и «перемножение» матриц.

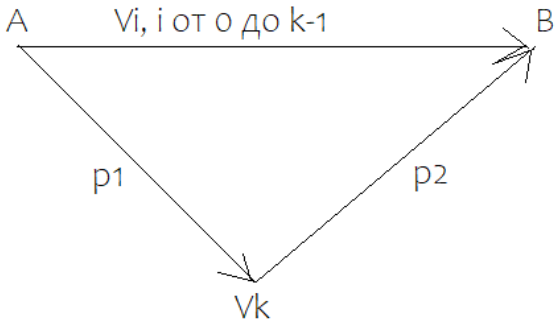
Что это?

30. Алгоритм Флойда-Уоршалла

Ключ: ДП, таблицу заполняем тем, что есть. Смотрим вершины по очереди от 0 до конечной, какие пути проходят через вершину? Можно ли их сократить (по типу релаксации)? $O(V^3)$. Проверка отрицательных циклов – ещё раз запустить.

Чистое динамическое программирование.

Пусть у нас есть кратчайшие пути для вершин от 0-й (A) до k-1-й (B). Где-то у нас существует вершина с номером k.



Тогда мы можем попасть из A в B либо по нашему пути от 0-й до k-1-й вершины (пусть будет путь S), либо через вершину с номером k (до неё путь P_1, после – P_2).

Тогда есть три варианта:

1) $S \le P_1 + P_2$, тогда путь и так оптимальный, мы идём из A в B, где вершины только от 0-й до k-1-й. Мы говорим, что можем заявить, что это то же самое, что и путь от 0-й до k-й вершины.
2) $S > P_1 + P_2$, путь неоптимальный, это будет путь от 0-й до k-й вершины.

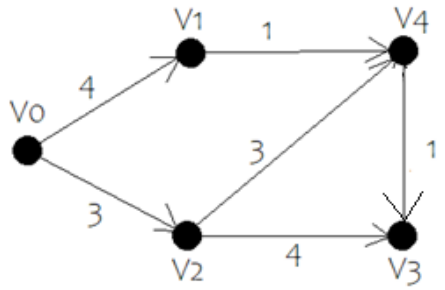
В чём у нас ДП: у нас есть все вершины, а давайте найдём все пути, кроме вершины, кроме V_n, а потом кроме вершины V_{n-1} и так далее, сокращаем и сокращаем - нисходящий анализ. Но нам нужен восходящий анализ. Поэтому, что мы ищем: все кратчайшие пути, которые содержат вершину 0. Далее все кратчайшие пути, которые содержат вершины 0 и 1. Далее все кратчайшие пути, которые содержат вершины 0, 1 и 2. И так далее.

Псевдокод:

```
FW(G):

For i = 0 to V-1:
        For j = 0 to V-1:
                d_ij^0 = d_ij' = ∞
        d_ii^0 = 0
For k = 0 to V-1:
        For i = 0 to V-1:
                For j = 0 to V-1:
                        If(d_ij^(k+1) > d_ik^k + d_jk^k)
                                d_ij^(k+1) = d_ik^k + d_jk^k
                        else
                                d_ij^(k+1) = d_ij^k
return d;
```

Пример:



Создаём матрицу:

	V_0	V_1	V_2	V_3	V_4
V_0					
V_1					
V_2					
V_3					
V_4					

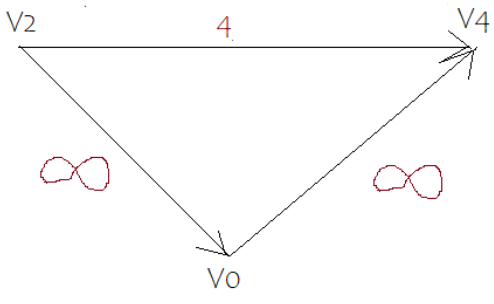
1) По диагонали кладём нули.

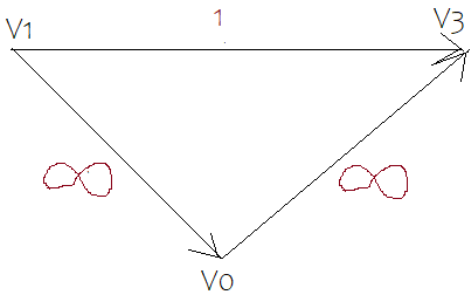
	V_0	V_1	V_2	V_3	V_4
V_0	0				
V_1		0			
V_2			0		
V_3				0	
V_4					0

2) Если существует ребро, ставим его. Горизонталь – откуда, вертикаль – куда. Значения, которых нет - ∞.

	V_0	V_1	V_2	V_3	V_4
V_0	0	4	3	∞	∞
V_1	∞	0	∞	1	∞
V_2	∞	∞	0	3	4
V_3	∞	∞	∞	0	1
V_4	∞	∞	∞	∞	0

3) Идём по алгоритму. k = 0. Сначала мы в качестве «обходной» вершины используем V_0. Смотрим: какие пути проходят через V_0? V_2-V_4 и V_1-V_3. Пытаемся улучшить веса. Смотрим по матрице пути: можем ли мы пойти, например, из V_2 в V_4 и V_2 в V_4 через V_0.(из V_2 в V_0 + из V_0 в V_4). 4 лучше, чем бесконечность, ничего не меняется. Для всех остальных рёбер, например, V_1 в V_3, улучшить тоже нельзя.





	V_0	V_1	V_2	V_3	V_4

V_0	0	4	3	∞	∞
V_1	∞	0	∞	1	∞
V_2	∞	∞	0	3	4
V_3	∞	∞	∞	0	1
V_4	∞	∞	∞	∞	0

4) В качестве «обходной» вершины используем V_1. Пути: V_0-V_3, V_0-V_4. Смотрим: можем ли мы пойти, например, из V_0 в V_3 и V_o в V_3 через V_1.. 4 + 1 лучше, чем бесконечность поэтому пишем 5. Для V_0 в V_4 улучшить нельзя, всё.

	V_0	V_1	V_2	V_3	V_4
V_0	0	4	3	5	∞
V_1	∞	0	∞	1	∞
V_2	∞	∞	0	3	4
V_3	∞	∞	∞	0	1
V_4	∞	∞	∞	∞	0

5) В качестве «обходной» вершины используем V_2. Пути: V_0-V_4, V_0-V_3. Смотрим: можем ли мы пойти из V_0 в V_3 и V_o в V_3 через V_2.. 4 + 1 лучше, чем бесконечность поэтому пишем 5. Для V_0 в V_4 улучшить нельзя, всё.

	V_0	V_1	V_2	V_3	V_4
V_0	0	4	3	5	7
V_1	∞	0	∞	1	∞
V_2	∞	∞	0	3	4
V_3	∞	∞	∞	0	1
V_4	∞	∞	∞	∞	0

И так далее.

Сложность: $O(V^3)$.

Наличие отрицательных циклов проверяется так: если мы уже прошли по всем вспомогательным вершинам, запустили для какой-то вершины и значение кратчайшего расстояния уменьшилось – есть отрицательный цикл.

31. Алгоритм Джонсона.

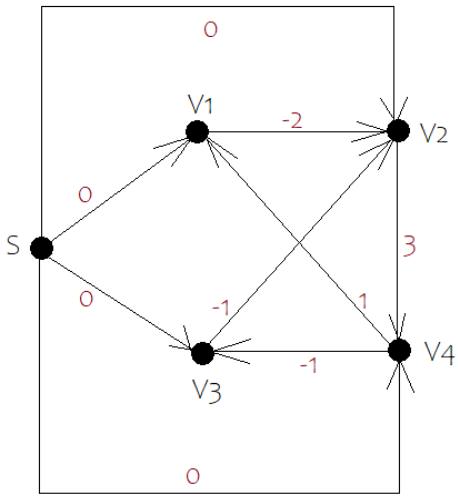
Ключ: для отрицательных рёбер. Создаём вершину с путями 0 к каждой. БФ для проверки отрицательных циклов. Перевзвешиваем все вершины $W'(V_i, V_j) = W(V_i, V_j) + d(S, V_i) – d(S, V_j)$, потом V раз Дейкстра, все пути, кроме циклов, одинаково меняются – на тот же вес. БФ + V раз Дейкстра. O(VE) + V * O((V+E)logV) = O(VE + V *(V+E)logV).

Он может работать с отрицательными рёбрами.

Алгоритм:

0) Создаём вершину S и соединяем её вместе с каждой вершиной графа с путями 0.
1) Далее запускаем алгоритм Белмана-Форда от нашего графа и вершины S - ищем наличие отрицательных циклов. Если он найдёт отрицательный цикл, останавливаемся.
2) Далее делаем перевзвешивание. $W'(V_i, V_j) = W(V_i, V_j) + d(S, V_i) – d(S, V_j)$, d – кратчайший путь.
3) После этого V раз запускаем алгоритм Дейкстры.
4) Кратчайший путь между i, j $P'(V_i, V_j) = P(V_i, V_j) + d(S, V_i) – d(S, V_j)$.

Пример:



Веса:

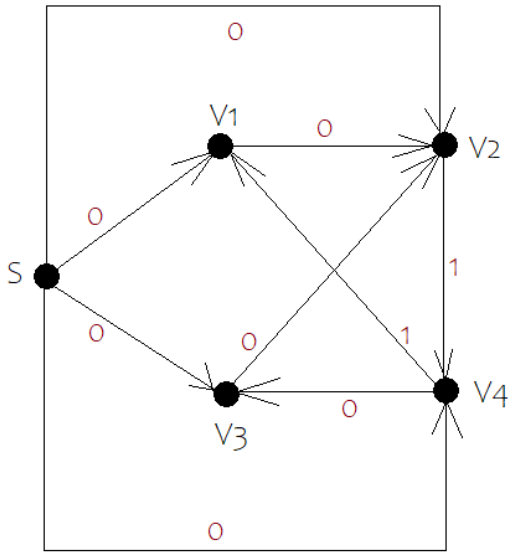
$W(V_1, V_2) = -2$
$W(V_2, V_4) = 3$
$W(V_4, V_1) = 1$
$W(V_3, V_2) = -1$
$W(V_4, V_3) = -1$

Допустим, мы запустили Белмана-Форда для S.

$d(S, V_1) = 0$
$d(S, V_2) = -2$
$d(S, V_3) = -1$
$d(S, V_4) = 0.$
Теперь мы поняли, что у нас нет отрицательных циклов.

Далее перевзвешиваем.

$W'(V_1, V_2) = W(V_1, V_2) + d(S, V_1) – d(S, V_2) = -2 + 0 – (-2) = 0.$
$W'(V_2, V_4) = 3 + (-2) – 0 = 1$
$W'(V_4, V_1) = 1 + 0 – 0 = 1$
$W'(V_3, V_2) = -1 + (-1) – (-2) = 0$
$W'(V_4, V_3) = -1 + 0 – (-1) = 0$



Далее найдём кратчайший путь между V_1, V_4.

$P(V_1, V_4) = 1,$
$P'(V_1, V_4) = P(V_1, V_4) - d(S, V_4) – d(S, V_1) = 1.$

Далее найдём кратчайший путь между V_1, V_3.

$P(V_1, V_3) = 1,$
$P'(V_1, V_3) = P(V_1, V_3) - d(S, V_3) – d(S, V_1) = 1 + 0 – 1 = 0.$

Почему это работает?

$V_i -> V_j -> V_k.$

$P(V_i -> V_k) = W(V_i, V_j) + W(V_j, V_k).$

$P'(V_i \to V_k) = W'(V_i, V_j) + W'(V_j, V_k) = W(V_i, V_j) + d(S, V_i) - d(S, V_j) + W(V_j, V_k) + d(S, V_j) - d(S, V_k) = W(V_i, V_j) + W(V_j, V_k) + d(S, V_i) - d(S, V_k) = P(V_j \to V_k) + d(S, V_i) - d(S, V_k)$.

Что это значит? Все пути совершенно одинаково меняются – на тот же вес. Кроме циклов всё будет одинаково меняться.

Сложность: БФ + V раз Дейкстра. $O(VE) + V * O((V+E)\log V) = O(VE + V *(V+E)\log V)$.

Если граф разреженный (мало рёбер), то $E \sim V$, то $O(V^2 * \log V)$

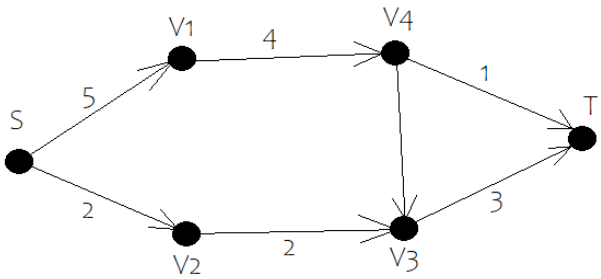
32. Задача о максимальном потоке. Алгоритм Форда-Фалкерсона. Алгоритм Эдмондса-Карпа.

Ключ: ориентированный нагруженный граф, источник, сток, поток, остаточная сеть, условие нагруженности, запускаем ФФ, когда пути закончились, конец. O(|F|*E), F – величина потока, E – количество рёбер

Постановка задачи о нахождении максимального потока

Есть ориентированный нагруженный граф, у которого есть две вершины, называющиеся источником и стоком.

Пусть есть граф, в котором S - источник, T - сток. Из источника мы можем запустить поток такой, что он не превышает вес ребра, которому мы его пускаем. Граф:

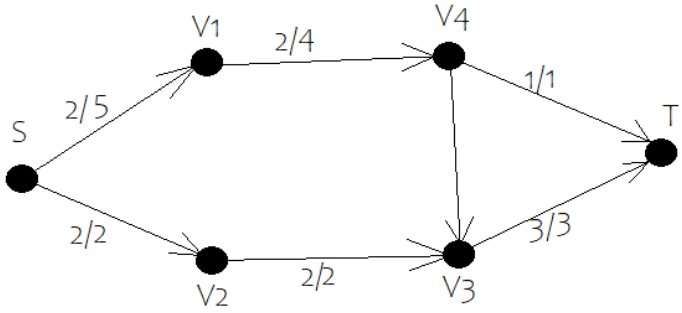


Мы можем пустить поток величины до 5 по первому ребру или до 2 по второму. Однако после прохождением потоком одного ребра, он пойдёт через второе.

Пусть поток величины 5 пустили через ребро $S\text{-}V_1$ с весом 5. Тогда он дальше пойдёт по ребру $V_1\text{-}V_4$, максимальная нагрузка которого – 4. Поток величины 5-4=1 застрянет в вершине V_1, а это недопустимо.

Однако мы можем пустить поток величины 1 через рёбра $S\text{-}V_1\text{-}V_4\text{-}T$ и загруженность рёбер не будет нарушена. Тогда мы получим поток величины три.

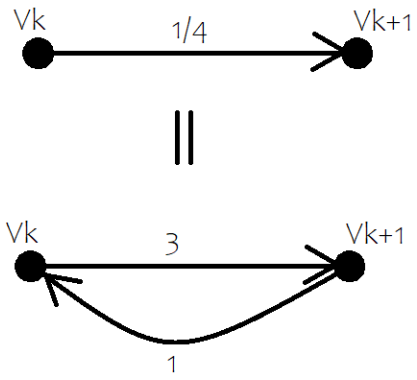
Поток – это суммарный путь по рёбрам из источника в сток при отсутствии нарушений условий о загруженности рёбер.



В данном графе он равен 4.

Сам алгоритм Форда-Фалкерсона основан на понятии остаточной сети:

Остаточная сеть – преобразование, при котором после запуска потока мы создаём обратное ребро и перевзвешиваем рёбра. По обратному пускаем значение потока, который мы запустили, а по искомому – то, которое мы ещё можем запустить.

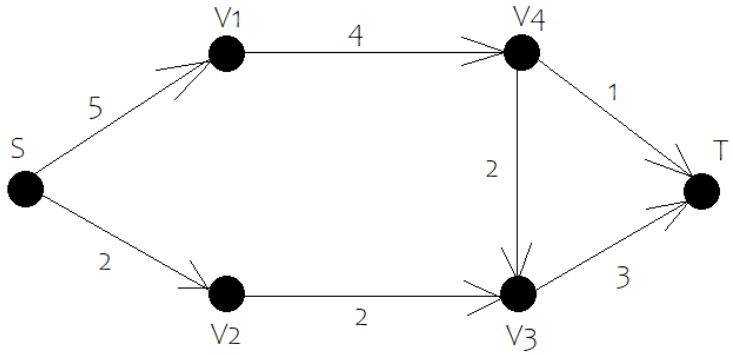


Когда из S в T пути закончились, мы говорим, что все потоки нашли.

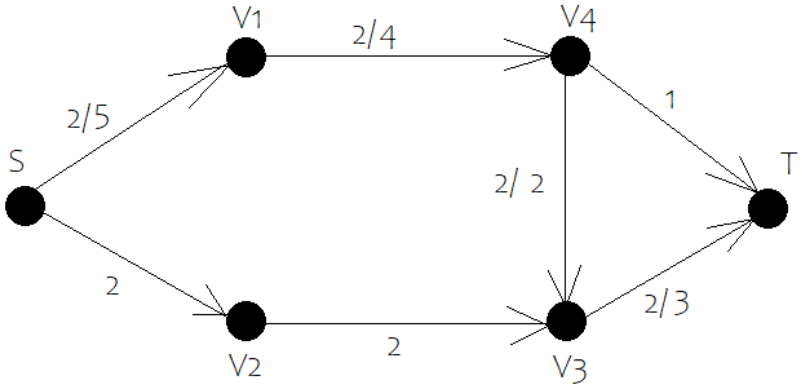
Пример алгоритма:

Решение через Форда-Фалкерсона

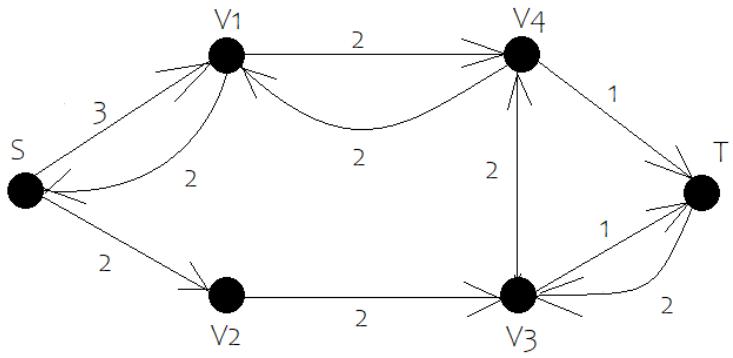
1) Сначала граф совпадает с остаточной сетью, так как запущенный поток 0.



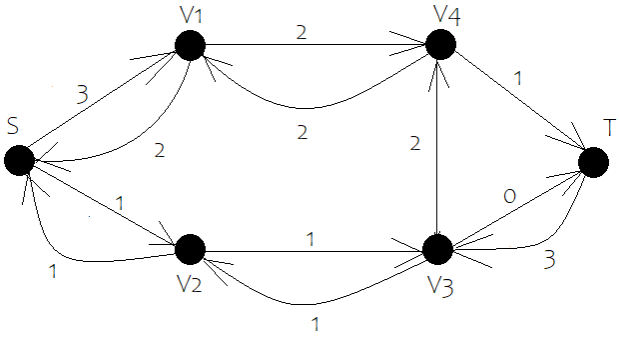
2) Что делаем: выбираем произвольный любой путь и пускаем поток.
Пусть будет путь S-V_1-V_4-V_3-T. Веса рёбер: 5, 4, 2, 3. Минимальный вес – 2. Значит, максимальный поток по этому пути – 2. Запускаем.



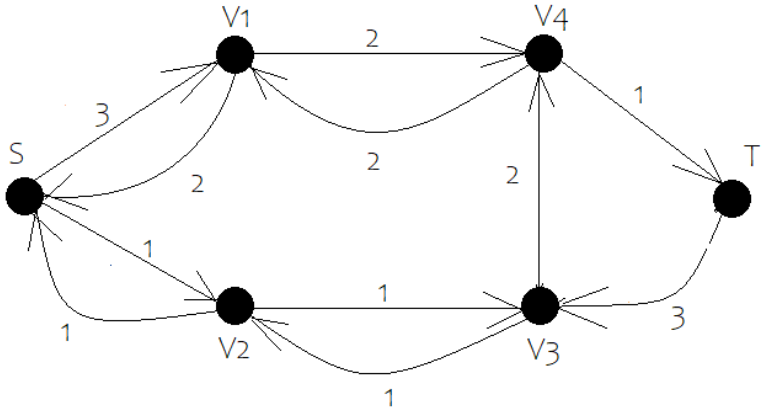
3) Рисуем остаточную сеть.



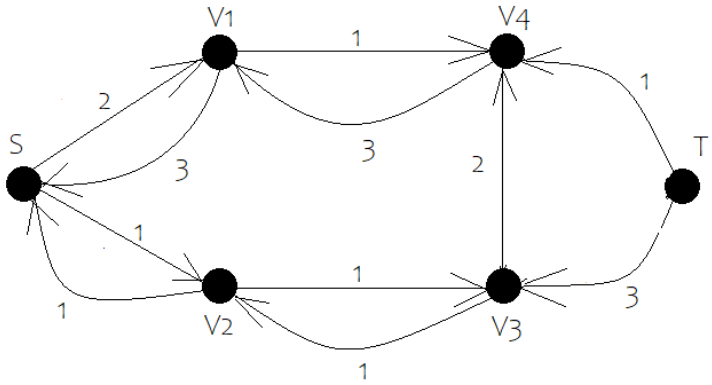
4) Ищем любой другой новый путь. Пусть будет путь S-V_2-V_3-T. Веса рёбер: 2, 2, 1. Минимальный вес – 1. Значит, максимальный поток по этому пути – 1. Суммарный поток – 2+1 = 3. Запускаем. Строим остаточную сеть.



При этом ребро весом 0 можно удалить.



5) Опять ищем путь. S-V_1-V_4-T. Веса рёбер: 3, 2, 1. Минимальный вес – 1. Значит, максимальный поток по этому пути – 1. Суммарный поток – 2+1+1 = 4. Запускаем. Строим остаточную сеть.



6) Есть ли ещё пути в T? Нет. Значит, алгоритм окончен. Максимальный поток 4.

Обход можно делать в ширину или глубину для поиска путей.

Сложность в худшем случае O(|F|*E), F – величина потока, E – количество рёбер.

А что делать, если несколько истоков и стоков?

Заведём новую вершину S – супер-источник, T – супер-сток. Соединим все стоки с супер-стоком и все источники с супер-источников. Вес соединительных рёбер – бесконечность. И действуем по алгоритму.

Алгоритм Эдмондса — Карпа — это вариант алгоритма Форда — Фалкерсона, при котором на каждом шаге выбирают *кратчайший* дополняющий путь из в остаточной сети (полагая, что каждое ребро имеет единичную длину). Кратчайший путь находится поиском в ширину.

сложность алгоритма Эдмондса — Карпа равна O(V*E^2)

33. Задача поиска максимального паросочетания в двудольном графе. Применение алгоритма Форда-Фалкерсона для поиска максимального паросочетания. Алгоритм Куна.

Ключ: двудольный граф: объединение - граф, пересечение – пустое множество; маркируем долями, перешли – сменили, если попали в ту же – не двудольный. Ф-Ф: исток – левые(кто), сток – правые (с кем). Пути ∞ 1 ∞. Ответ искомый. Кун: 0 – принадлежит паросочетанию, 1 – нет. Чередующиеся – 0 1 чередуются, увеличивающий – (0-***-0). Если нет увеличивающих путей, нашли ответ. Если вершина была уже с кем-то соединена, то 1 и идём в неё, если нет – 0 и идём дальше(либо из вершины, либо, если нет путей, в новую неиспользованную).

Отступление. Что такое двудольный граф?

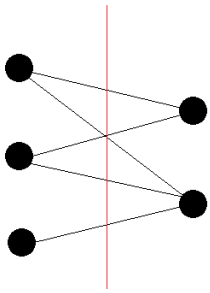
Это граф и множества вершин и рёбер такой, что его можно разбить на два подмножества так, что объединение этих множеств даёт исходный граф, а пересечение – пустое множество.

G = (V, E)

V_1 и V_2 = ø

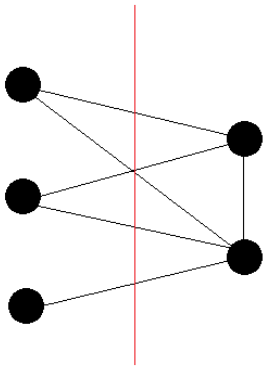
V_1 или V_2 = V

<v, u> ϵ E : v ϵ V_1, u ϵ V_2 || v ϵ V_2, u ϵ V_1



Его условно можно разрезать на две части, все рёбра будут разрезаны.

НЕ двудольный граф:



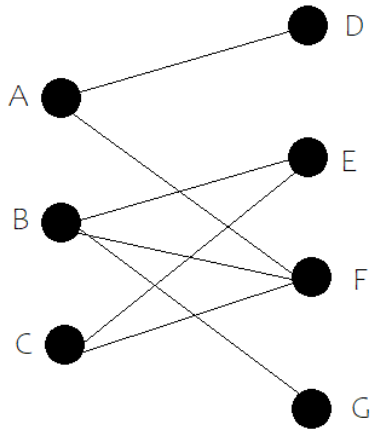
Как определить, является ли граф двудольным? Запускаем от какой-то вершины обход с маркировкой по отношению к долям. Вершину один маркируем первой долей, перешли по ребру – вторую вершину маркируем второй долей и так далее. Если мы попали из второй доли во вторую или из первой в первую – граф не двудольный.

Задача: поиск максимального числа пара-сочетаний.

Можно решить двумя способами, первый – методом Форда-Фалкерсона, второй – алгоритм Куна.

Дано:

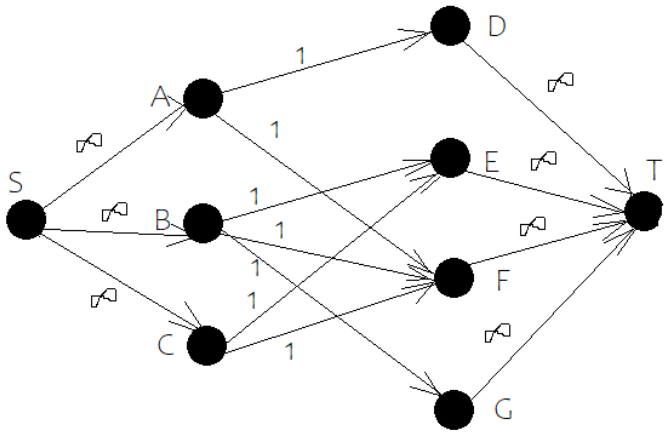
Пусть дотеры разделились на две группы и хотят сыграть в доту один на один. A хочет сыграть с D, а можно A ещё и с F. B же хочет сыграть с E, F или G. C с E или F.



Нужно найти максимальное паросочетание. Что это? Предположим, A стал играть с F, B с E. На этом пары закончились. Мы получили два паросочетания. A могли A играть с D, C с F, B с E. Это уже три паросочетание, также это максимальное количество паросочетаний для данной модели. Умным языком: надо найти максимальное количество рёбер так, что каждой вершине инцидентно не более одно ребро.

Решение с помощью алгоритма Форда-Фалкерсона:

Все рёбра направленные, их вес – единица. Заводим дополнительно исток и сток, соединяем их соответственно с левыми и правыми вершинами.



И тупо решаем Форда-Фалкерсона. Величина потока и будет ответом.

Решение с помощью алгоритма Куна

В чём идея: маркируем рёбра 0 или 1, 0 – не принадлежит нашему паросочетанию, 1 – принадлежит. У нас может быть какой-то путь, в котором могут рёбра как принадлежать паросочетанию, так и нет. Например, есть путь V_1-V_2-V_3-V_4-V_5(**где вершины чередуются из разных долей – ОБЯЗАТЕЛЬНО**), маркировки рёбер 1-0-1-0. Или V_1-V_2-V_3-V_4, 0-1-0.

Если значения принадлежности чередуются, то пути называются **чередующимися.**

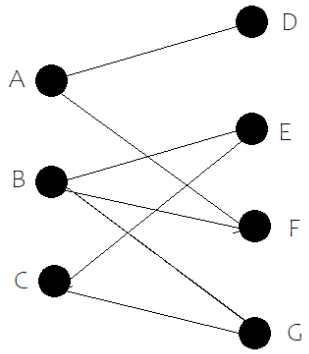
Увеличивающий путь – тот, который начинается и кончается не принадлежащим рёбрам. 0-1-0.

Теорема Бёржа:

> Если в графе больше нет увеличивающих путей, то мы нашли максимальное паросочетание.

Как искать? Мы берём произвольную вершину и переходим по одному из её рёбер. Так как эта вершина раньше не использовалась, смело по ней переходим. Переходим, смотрим: связано ли это ребро уже с какой-то другой вершиной? Если нет, добавляем ребро и забываем про него, если связано, переходим по ребру и ищем другую какую-то вершину, с которой оно связано.

Пример:



1) С чем соединена A? D, F. Идём, например, в F. F соединено с кем-то? Нет. Наше текущее паросочетание A-F, 0.
2) Окей. Смотрим следующую вершину – B. Она соединена с E, F, G. Допустим, мы пошли в F. F соединено с кем-то? Да, с A. Поэтому F-A, 1. Получаем путь B-F-A, 0-1. Идём в A.
3) Куда можем пойти из A? B D. D не обрабатывали, A-D, 0. Итоговый путь B-F-A-D, 0-1-0.
4) Выкинем рёбра, отмеченные 1, выкинем, а рёбра, отмеченные 0, добавляем. То есть остаются элементы паросочетаний B-F, A-D.
5) Откуда мы ещё не ходили? C. Идём из C в E, добавляем к множеству. B-F, A-D, C-E. 0-0-0. Чередований нет – мы нашли максимальное паросочетание.

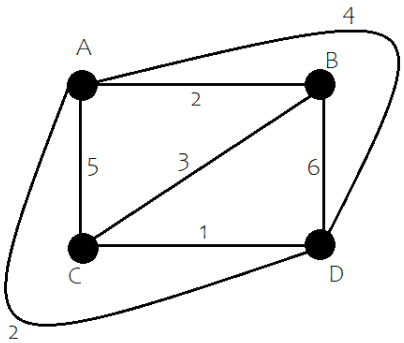
34. Минимальные остовные деревья. Алгоритмы Крускала и Прима.

Ключ: Крускало: ЖА, минимально возможное ребро с условием добавляем в дерево, после выкидываем ненужные, набрали V-1 ребро – стоп. Условие: объединять можно вершины из разного множества, как только объединили – идут в одно. Сложность: O(E*logV). Прима: Дейкстра, только после каждой релаксации сортируем очередь, чтобы взять вершину, которая достижима быстрее всего.

Алгоритм Крускало

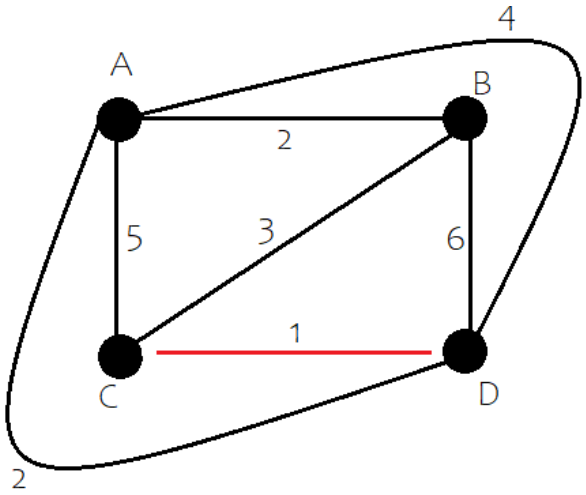
Его идея в том, чтобы попробовать решить задачу жадным алгоритмом и каждый раз брать минимально возможное ребро, которое выполняет условие. И добавляем ребро в наше остовное дерево. После этого выкидываем рёбра, которые нам не нужны. Когда набрали V-1 ребро, останавливаемся. Условие: представим, что все вершины – отдельные множества. Условие: мы можем соединить вершины, только если они – разные множества. После их соединения, множества, в которое вершины входят, объединяются. Мы берём то ребро с минимальным весом, чьи вершины принадлежат разным множествам.

Пример:



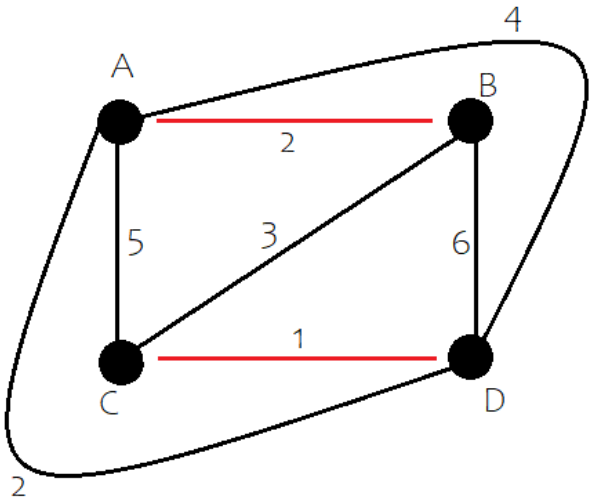
1) Пока все вершины принадлежат своим множествам:
{A}, {B}, {C}, {D}.
Ищем минимальное ребро. C-D = 1. Добавляем это ребро в основное дерево, а множества C и D объединяем.

Вес 1.



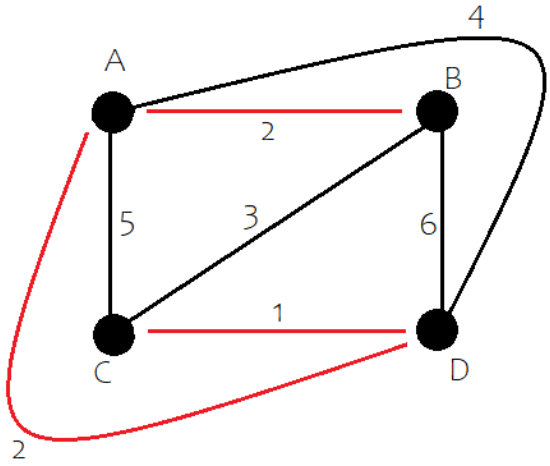
Имеем: {A}, {B}, {C, D}.

2) Смотрим минимальное ребро. Их несколько. Пусть будет A-B. A, B лежат в разных множествах. Вес 1 + 2 = 3.



Имеем {A, B}, {C, D}.

3) Смотрим следующее минимальное ребро. Пусть будет A-D. A, D лежат в разных множествах. Вес 1 + 2 + 2 = 5



Имеем {A, B, C, D}. Конец. Итоговый вес – 5.

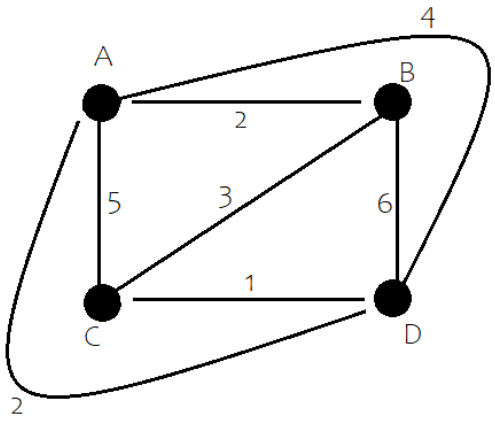
Сложность: O(E*logV)

Алгоритм Прима

Идея:

«Почти Дейкстра». Мы рисуем граф, выбираем произвольно вершину, считаем, что она достижима за 0, остальные за бесконечность. На каждом шаге делаем следующее: вынимаем самую маленькую вершину из своего множества (сначала наша, потом – по достижимости), а затем, для всех рёбер, выходящих из неё, делаем релаксацию. Снова ищем вершину, которая достижима быстрее всего, берём её, потом для всех рёбер из неё делаем релаксацию и т п, пока множество вершин не закончится.

Пример:



1) Начинаем с любой вершины. Пусть C. Тогда

Вершина	C	A	B	D
Достижимость	0	∞	∞	∞

Вычёркиваем C. Для всех рёбер, выходящий из неё, делаем релаксацию.

Вершина	C	A	B	D
Достижимость	0	5	3	1

Сортируем:

Вершина	D	B	A
Достижимость	1	3	5

2) Берём D.

Вершина	D	B	A
Достижимость	1	3	5

Релаксируем.

Вершина	B	A
Достижимость	3	2

Сортируем.

Вершина	A	B
Достижимость	2	3

3) Вершина A.

Вершина	A	B
Достижимость	2	3

Релаксируем.

Вершина	B
Достижимость	2

4) Осталась только B. Конец.

Блок IV. Прочие темы – Полиномы, Фурье

Быстрое преобразование Фурье

Что такое полином? Это функция в виде $p(x) = a_0 + a_1 x + a_2 x^2 + \dots + a_n x^n = \sum_{i=0}^{n} a_i x^i$.

$a_n \neq 0$, n – это степень полинома.

Хранить полином можно так (коэффициентная форма):

a_0	a_1	…	a_n
0	1	…	n

Три вида действий с полиномами:

1) Значение в точке x
2) Сумма полиномов
3) Произведение полиномов

Итак:

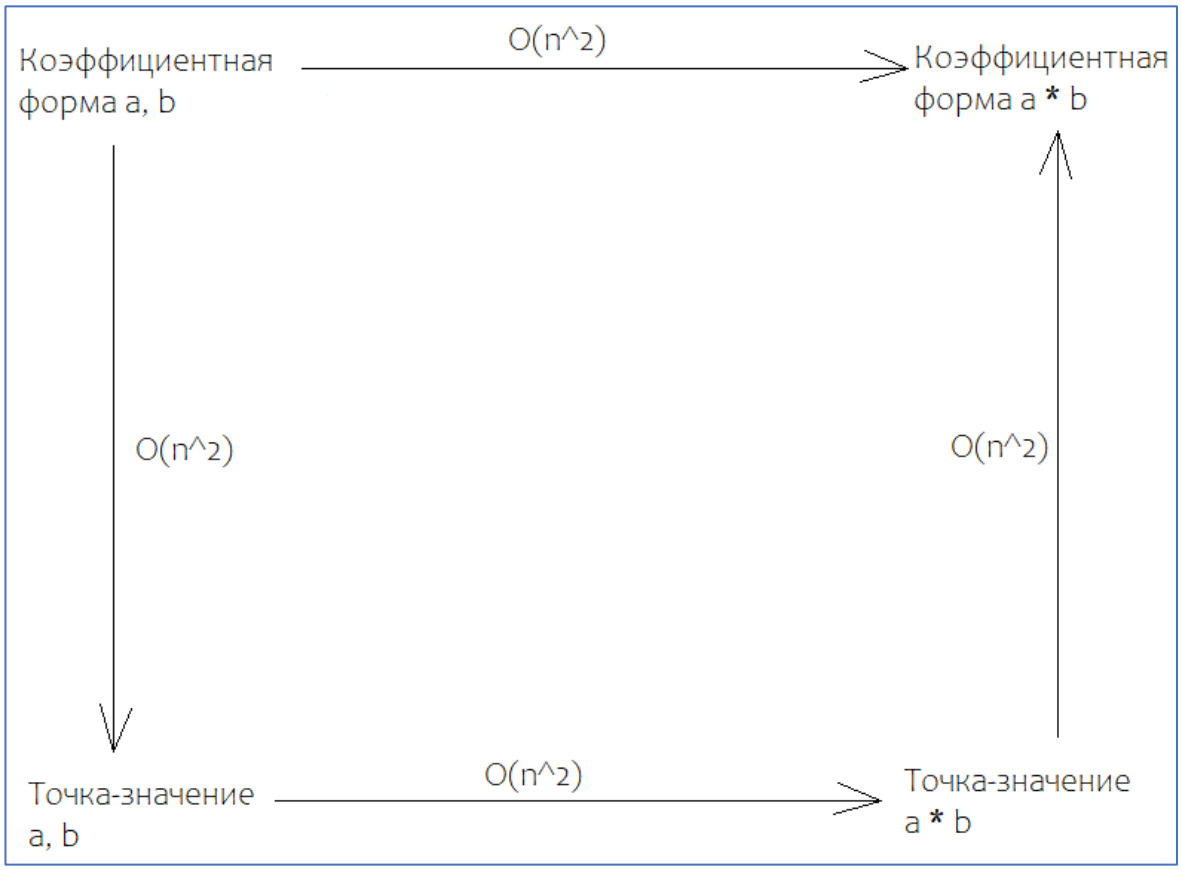
1. Как посчитать значение в точке x:
   Если считать отдельно по x^i $p(x) = a_0 + a_1 x + a_2 x^2 + \dots + a_n x^n$, то будет сложность $\sum_{i=0}^{n} i \sim n^2$.
   Если посчитаем так $p(x) = (((a_n x + a_{n-1}) x + a_{n-2}) x + \dots)$, то сложность будет O(n) – схема Горнера.
2. Как складывать:
   $a(x) = \sum_{i=0}^{n} a_i x^i$, $b(x) = \sum_{j=0}^{m} b_j x^j$,
   $c(x) = a(x) + b(x) = \sum_{k=0}^{\max(n,m)} (a_k + b_k) x^k$.
   Сложность вычислений: O(max(n, m)).
3. Произведение:
   $a(x) = \sum_{i=0}^{n} a_i x^i$, $b(x) = \sum_{j=0}^{m} b_j x^j$,
   $c(x) = a(x) * b(x) = \sum_{k=0}^{n+m} c_k x^k$.
   $c_k = \sum_{i=0}^{k} a_i * b_{k-i}$.
   Сложность вычислений: O(n * m).

Способ 2 хранения полинома:

С помощью точек – как пару (x_i, y_i) от 0 до n-1, где $y_i = p(x_i)$. Ограничим количество точек меньше или равно 2n-1.

1. Как посчитать значение в точке x:
   Это просто наше y из пары. Если его нет, то ничего не можем сделать, иначе за константу даём ответ.
2. Сумма:
   c(x) = a(x) + b(x), просто складываем два значения за линейное время.
3. Произведение – аналогично, два числа перемножаем – за линейное время. c(x) = a(x) * b(x).

Полиномы и их формы.



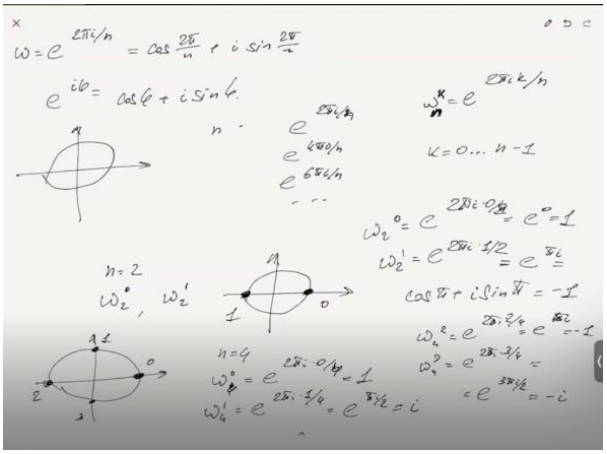
Из точки-значения a*b мы можем получить коэффициентную форму a*b, если распишем полином в коэффициентной форме для каждой точки и решим эту систему через матрицы ($X*a^T = y^T$), решается через определитель Вандермонда, который работает за квадратичное время.

Тогда встаёт вопрос: зачем нужен сложный путь через точку-значение, если и так, и так за квадрат работает? Другой разговор, если преобразование между точкой-значением везде будет O(nlogn). Вот тут и появляются преобразования Фурье.

Есть формула Эйлера:

$W = e^{2\pi i/n} = \cos(2\pi/n) + i \sin(2\pi/n)$.

$W^g_n = e^{2\pi i g/n}$



n говорит, на сколько точек мы разобьём нашу окружность на комплексной плоскости. Каждой точке соответствует какая-то противоположная.

Идея:

Есть p(x) = $\sum_{j=0}^{n-1} a_j x^j$.

Мы можем посчитать для него $w^0{}_n$, $w^1{}_n$, … $w^{n-1}{}_n$.

$y_k = p(x_k) = \sum_{j=0}^{n-1} a_j w_n^{kj}$ – это дискретное преобразование Фурье. То есть вместо того, чтобы считать в вещественных числах, мы переводим всё в комплексные.

Как его использовать?

Пример: $p(x) = a_0 + a_1 x + a_2 x^2 + \dots + a_{n-1} x^{n-1}$

Давайте полином разобьём на два: с чётными - g(x) и нечётными – h(x) коэффициентами.

$g(x) = a_0 + a_2 x^2 + a_4 x^4 + a_6 x^6 + \dots + a_{n-1} x^{n-1} = = a_0 + a_2 (x)^2 + a_4 (x^2)^2 + a_6 (x^3)^2 + \dots + a_{n-2} (x^{(n-2)/2})^2$

$h(x) = x * (a_1 + a_3 (x)^2 + a_5 (x^2)^2 + a_7 (x^3)^2 + \dots + a_{n-1} (x^{(n-2)/2})^2)$

$p(x) = g(x) + h(x) = A_0(x^2) + x * A_1(x^2)$

$A_0 = (a_0, a_2, \dots a_{n-2})$

$A_1 = (a_1, a_3, \dots a_{n-1})$

Оба полинома считаются в точке x^2. X у нас n.

$X = w^0{}_n, w^1{}_n, \dots w^{n-1}{}_n$.

Как посчитать w?

Возьмём $(W^g{}_n)^2 = e^{4\pi i g/n}$ – для x_g.

Возьмём $(W^{g+n/2}{}_n)^2 = e^{4\pi i g/n} * e^{2\pi i}$ – для $x_{g+n/2}$. = $(- W^g{}_n)^2$.

То есть, если есть какие-то две симметричные точки на окружности относительно центра, их квадрат – одна и та же точка.

То есть надо посчитать уже n/2 точек.

Пусть $A_0((W^g{}_n)^2) = X$, $A_1((W^g{}_n)^2) = Y$.

Быстрое преобразование Фурье:

$P(W^g{}_n) = X + Y * W^g{}_n$.

$P(W^{g+n/2}{}_n) = X - Y * W^g{}_n$.

Пример кода:

```
FFT(a):
    If a.size == 1:
        Return a.
    n = a.size
    w_n = e^(2πi/n)
    w = 1
    a[0] = { a_0, a_2, … a_{n-2}}
    a[1] = { a_1, a_3, … a_{n-1}}
```

y[0] = FFT(a[0])
y[1] = FFT(a[1])
for k = 0 to n/2
$y_k = y[0]_k + w * y[1]_k$
$y_{k+n/2} = y[0]_k - w * y[1]_k$
return $w = w_n$

Пример:

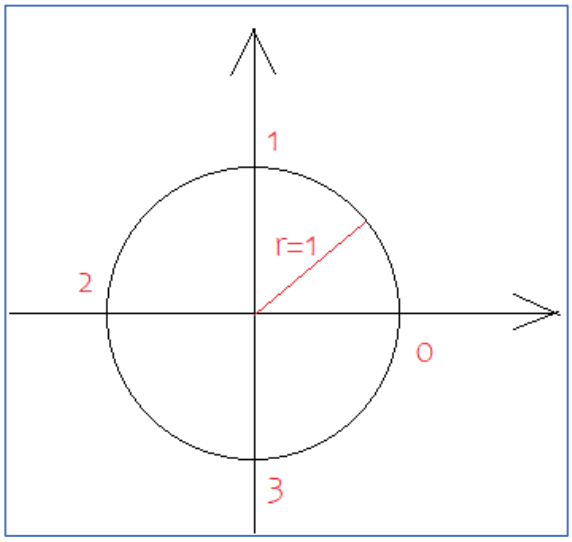
$a(x) = 2 + 2x + x^2$;

$b(x) = 3x$;

Что делаем:

1) Для $a(x) = 2 + 2x + x^2$;

Давайте зададим 4 точки на комплексной плоскости. Финальная степень полинома – не больше 4-х, поэтому. Тогда у нас есть w^0_4, w^1_4, w^2_4, w^3_4. Расположим на окружности по формуле Эйлера.



$a = [a_0, a_1, a_2, a_3] = [2, 2, 1, 0]$.

1. $a[0] = [a_0, a_2] = [2, 1]$, разделим полином ещё на две части: a[0] = 2, a[1] = 1. Решаем теперь задачу для каждого из них. Но решать её не надо, нужно просто посчитать по формуле то, что было + w * на то, что стало – запустили быстрое преобразование Фурье.
   y[0] = [2+1, 2-1] = [3, 1].
   Что это значит? a[0] = [2, 1] – полином 2x+1. При x = 1 получим 3, при x = -1 получим 1.
2. $a[1] = [a_1, a_3] = [2, 0]$, разделим полином ещё на две части: a[0] = 2, a[1] = 0. Решаем теперь задачу для каждого из них (так как уже по одному значению). Тут y[1] = [2+0, 2-0] = [2, 2].

Что делаем дальше?

$y_0 = y[0]_0 + 1 * y[1]_0 = 3 + 2 = 5$;

$y_2 = y[0]_0 - 1 * y[1]_0 = 3 - 2 = 1$;

$y_1 = y[0]_1 + i * y[1]_1 = 1 + 2i;$

$y_3 = y[0]_1 - i * y[1]_1 = 1 – 2i.$

$y = [5, 1+2i, 1, 1-2i].$

2) Для $b(x) = 3x;$

$b = [b_0, b_1, b_2, b_3] = [0, 3, 0, 0].$

1. $b[0] = [b_0, b_2] = [0, 0]$, разделим полином ещё на две части: $b[0] = 0$, $b[1] = 0$.
   Запустили быстрое преобразование Фурье.
   $y[0] = [0+0, 0-0] = [0, 0].$
2. $b[1] = [b_1, b_3] = [3, 0]$, разделим полином ещё на две части: $b[0] = 3$, $b[1] = 0$.
   Запустили быстрое преобразование Фурье
   Тут $y[1] = [3+0, 3-0] = [3, 3].$

$y_0 = 0 + 1 * 3 = 3.$
$y_2 = 0 – 1 * 3 = -3.$
$y_1 = 0 + i * 3 = 3i.$
$Y_3 = 0 – i * 3 = -3i.$

$y = [3, -3, 3i, -3i].$

3) Теперь есть два вектора. $y_1 = [5, 1+2i, 1, 1-2i]$. $y_2 = [3, -3, 3i, -3i]$.
   Просто всё перемножаем.
   $y_{res} = [5*3, -3*(1+2i), 3i*1, -3i*(1-2i)] = [15, 3i-6, -3, -3i-6].$

   То есть $a(x)*b(x) = 6x + 6x^2 + 3x^3$.
   При $x = 1, -1, i, -i$ получим все значения y_{res}.
4) Для обратного дискретного преобразования Фурье есть формула:
   $a_j = \frac{1}{n}\sum_{k=0}^{n-1} y_k w_n^{-ki}$

Ключи

Рюкзак	**Конвейер**	**Перемножение матриц**
Ценность, сортируем по убыванию, дискретная не решается с помощью ЖА, контрпример.	равноправные этапы, два массива, подзадачи элементарны и зависимы, на каждой итерации выбираем минимальный возможный путь - оптимизация, $\min(S_{I,i-1}, S_{II,i-1} + P_{i-1}) + a_i$, O(2n). В лоб – 2^n.	числа Каталана, сложность перемножения двух матриц - p * q * r, если конечный вариант оптимален, его подзадачи тоже оптимальны, считаем самые маленькие подзадачи (сначала * двух матриц, потом трёх и доходим до ответа), сложность $O(n^3)$ для 4-х матриц.
LCS $LCS(X_1X_2…X_n, Y_1Y_2…Y_m)$ = { при $X_n = Y_m$ 1+ $LCS(X_1X_2…X_{n-1}, Y_1Y_2…Y_{m-1})$, при $X_n \neq Y_m$ $\max(LCS(X, Y_1Y_2…Y_{m-1}), LCS(X_1X_2…X_{n-1}, Y))$. при m = 0 || n = 0 0 } Таблица: если =, то диагональ + 1, если нет – то максимум из верхней и нижней. O(n*m).	**LD Левенштейн** Замена, вставка, удаление. $LD(X_1X_2…X_n, Y_1Y_2…Y_m)$ = { при $X_n = Y_m$ $LD(X_1X_2…X_{n-1}, Y_1Y_2…Y_{m-1})$, при $X_n \neq Y_m$ $\min(LD(X, Y_1Y_2…Y_{m-1})$+insert, $LD(X_1X_2…X_{n-1}, Y)$+del, $LD(X_1X_2…X_{n-1}, Y_1Y_2…Y_{m-1})$+repl). при n = m = 0 0 при m = 0 delete * n при n = 0 insert * m } Таблица: если =, то диагональ. Если нет, то либо влево + удаление, либо вверх + вставка, либо диагональ + замена.	**LIS, LIS + LCS** LIS: покрытие, если > + новый список, <= кладём в максимально близкий список. Кол-во списков - ответ. ЖА, тк всегда невозрастание в списках. Восстановить – ссылка на последнее число. O(n * поиск= n, бинарный logn). LCS: Позиции букв 1 слова отсортируем, подставим последовательность вместо букв в 2, запустим LIS, кол-во списков – ответ, по ссылкам – индексы неотсортированные из 1. R*logR, длина 2 в цифрах, для длинных строк - ок
ДП **Подход,** на меньшие такие же подзадачи, могут пересекаться. использование накопленных данных, не 1 метод, парадигма. **Оптимизация.** Решение подзадач объединить и получить итог - рекуррентная структура. Можно вывернуть - из решения маленьких задач можно получить большое. Нет общих ресурсов. **Нисходящий** = Решаем от маленьких задач к большим. **Восходящий** = Разбить задачу на маленькие, решить их и из них составить большую. **Мемо**- оптимизация запоминания и устранении повторных расчётов	**Оптимальный выбор процессов** В лоб: M – игры от начала первой до конца второй, M' – множество M непересекающихся игр, максимальное количество элементов – ответ. ДП: какая-то игра гарантированно входит, тогда выбираем максимум из оптимального ответа до неё, после и неё – $O(n^3)$. ЖА: сортируем по концу - неубывание. Первый процесс – текущий, входит в ответ, потом процессы, начинающиеся раньше, чем первый кончился, пропускаем. Первый процесс, начинающийся позже, чем конец текущего, назовём текущим - O (n * сортировка).	**Хаффман** вариант за nlogn: сортируем по частоте в кучу, вытаскиваем два минимальных значения (каждый перекинули в конец, отсортировали), из вытащенных элементов строим узел дерева. Кладём узел в кучу. Потом вытащили ещё два элемента и тп. Частота суммируется. Обход: лево – 0, право – 1. Декодируем так же. На каждом шаге берём минимум – ЖА. За n: два массива сорт по возрастанию, минимум суммы двух редких из: первого, второго и первого, кладём во второй массив. По итогу получим дерево как в 1. Вычёркиваем по 2 элемента – O(n).

ЖА оптимальный ответ не в целом, а на текущем шаге, максимизировать ответ на каждом следующем шаге. Все подзадачи, кроме одной, пустые.	**В глубину** рекурсивно запускаем от непосещённого ребёнка, пока не закончились непосещённые или просто дети, тогда делаем шаг назад и опять рекурсия.	**В ширину** Фронт волны, очередь текущей обработки Q и список посещённых вершин used. Проходим, за исключением used. **O(v + e).** Очередь - пара: номер вершины и счётчик. Добавление смежных вершин +1.
Беллман-Форд путь начальной 0, остальные ∞. Релаксируем, пока можем, иначе ответ. V-1 раз, иначе отриц цикл. O(E * (V – 1) + E) = O(V * E), релаксируем E рёбер V-1 раз. E в худшем случае примерно равна V^2.	**Дейкстра** ЖА, не для отрицательных рёбер, строим очередь с приоритетом, путь начальной 0, остальные ∞. Удаляем вершину с минимальным весом, релаксируем все рёбра из неё и так далее. O(V^2 + E). На куче O(VlogV + ElogV). Если неполный граф, проще массив.	**Флойда-Уоршолла** ДП, таблицу заполняем тем, что есть. Смотрим вершины по очереди от 0 до конечной, какие пути проходят через вершину? Можно ли их сократить (по типу релаксации)? O(V^3). Проверка отрицательных циклов – ещё раз запустить.
Джонсона для отрицательных рёбер. Создаём вершину с путями 0 к каждой. БФ для проверки отрицательных циклов. Перевзвешиваем все вершины $W'(V_i, V_j) = W(V_i, V_j) + d(S, V_i) – d(S, V_j)$, потом V раз Дейкстра, все пути, кроме циклов, одинаково меняются – на тот же вес. БФ + V раз Дейкстра. O(VE) + V * O((V+E)logV) = O(VE + V *(V+E)logV).	**Форда-Фалкерсона** ориентированный нагруженный граф, источник, сток, поток, остаточная сеть, условие нагруженности, запускаем ФФ, когда пути закончились, конец. O(|F|*E), F – величина потока, E – количество рёбер	**Парасочетания, Ф-Ф** двудольный граф: объединение - граф, пересечение – пустое множество; маркируем долями, перешли – сменили, если попали в ту же – не двудольный. Ф-Ф: исток – левые(кто), сток – правые (с кем). Пути ∞ 1 ∞. Ответ искомый.
Парасочетания, Кун Кун: 0 – принадлежит паросочетанию, 1 – нет. Чередующиеся – 0 1 чередуются, увеличивающий – (0-***-0). Если нет увеличивающих путей, нашли ответ. Если вершина была уже с кем-то соединена, то 1 и идём в неё, если нет – 0 и идём дальше (либо из вершины, либо, если нет путей, в новую неиспользованную).	**Крускало** ЖА, минимально возможное ребро с условием добавляем в дерево, после выкидываем ненужные, набрали V-1 ребро – стоп. Условие: объединять можно вершины из разного множества, как только объединили – идут в одно. Сложность: O(E*logV).	**Прим** Дейкстра, только после каждой релаксации сортируем очередь, чтобы взять вершину, которая достижима быстрее всего.