**http://codeforces.ru/blog/entry/1712**

**Задача A: Подарок (Роман Едемский)**

Предположим, что оптимальным решением является пара (*A*, *B*), где *A* - количество золотых монет в подарке, а *B* - количество серебряных монет.  Нетрудно заметить, что существуют такие 2 индекса *i* и *j* (возможно совпадающих), что *gi* = *A* и *sj* = *B*, так как в противном случае мы бы могли уменьшить *A* или *B*, не нарушив связность графа.

Обозначим через R(A,B) граф, в котором для всех *i* выполняется *gi* ≤ *A* и  *si* ≤ *B*.

Обозначим через *T*(*A*) взвешенный граф, в котором для всех ребер выполняется ограничение *gi* ≤ *A*, а весами ребер будут *si*. Найдем в этом графе остовное дерево, у которого максимальное ребро - минимальное возможное. Можно показать, что для данного фиксированного *A* наименьшим значением *B*, при котором граф *R*(*A*, *B*) все еще связный, как раз будет это значение максимального ребра.

*Утверждение*. Минимальный остов графа одновременно является остовом, у которого максимальное ребро - минимальное возможное.

*Доказательство*. Рассмотрим любой минимальный остов L. Если существует остов P, у которого все ребра имеют строго меньший вес, чем вес максимального ребра у L. Тогда мы могли бы удалить из L макс. ребро и заменить его каким-то ребром из P, тем самым уменьшив его вес. Поскольку L - минимальный остов, мы не можем уменьшить его вес, следовательно такого P не существует.

Отсортируем все ребра исходного графа по возрастанию *gi*. Переберем все возможные значения *A* среди *gi*. Тогда ребрами графа *T*(*A*) для фиксированного *i* будут все ребра с индексами *j* ≤ *i*. Осталось научиться быстро находить мин. остов в этом графе.

Предположим, что мы для какого-то *i* нашли мин. остов в графе *T*(*gi*) (в случае, если граф несвязный, то в каждой его компоненте связности найден мин. остов). Добавим в него *i* + 1-е ребро. Если это ребро соединяет разные компоненты связности, просто добавим его, в противном случае в дереве образуется ровно один цикл. Найдем в нем максимальное ребро и удалим его из графа, получив таким образом новый минимальный остов в компоненте, куда добавилось ребро (доказательство опускается).

Поиск максимального ребра в цикле на дереве можно осуществить за *O*(*N*), а весь алгоритм будет работать за *O*(*NM* + *MlogM*).

**Задача B: Мыши и сыр (Роман Ризванов)**

Легко заметить, что количество ближайших кусков сыра для каждой мыши не превышает 2.

Найдем для каждой мыши ближайший сыр слева и справа. Из двух направлений выберем то, которое дает более короткий путь или оба, если длины путей одинаковы. Если на выбранном пути между мышью и сыром есть другая мышь, то мы это направление изымаем из рассмотрения, так как другая мышь быстрее съест тот сыр. Теперь все направления мышей непосредственно ведут к сыру и к каждому куску сыра ведет не более одного направления с каждой стороны.

Будем обрабатывать мышей слева направо. Если очередная мышь может двигаться влево и сыр слева не выбрала никакая из уже рассмотреных мышей или этот сыр выбрала мышь с таким же расстоянием до сыра, то текущая мышь двигаясь влево успеет поесть сыр, не мешая остальным мышам. Так как к сыру ведет не более одного направления, то этот выбор не повлияет на следующих мышей, а поэтому и не может в дальнейшем ухудшить ответ. В остальных случаях, двигаясь влево мы не сможем улучшить ответ, поэтому если мышь может двигаться вправо, то это единственная возможность для этой мыши поесть сыра, поэтому она выбирает движение вправо. Если мышь не может двигаться вправо, то она никак не сможит улучшить ответ.

Сложность решения *O*(*N* + *M*).

Также существует решение этой задачи при помощи динамического программирования.

**Задача C: Мутация (Ярослав Твердохлеб)**

В разборе вместо терминов "риск", "геном" и "ген" будут использоваться термины "стоимость", "строка" и "символ". Обозначим начальную строку через *S*.

Обозначим через *M* битовую маску символов, которые будут удалены. Переберем все возможные *M*. Для фиксированного *M* найдем стоимость строки, которая получилась и если она не превышает *T* - увеличим ответ на 1. Реализация этого алгоритма "в лоб" работает за *O*(2*KN*).

Избавиться от перебора нам не удастся, поэтому будем улучшать нахождение стоимости строки для фиксированного *M*.

Рассмотрим некоторую пару индексов символов из начальной строки *l* и *r*. Обозначим через *M*' битовую маску всех символов, которые лежат строго между ними. Если http://codeforces.ru/renderer/a559745e16bffa8c0c3fc965c8ccac3c9977bedf.pngили http://codeforces.ru/renderer/201c436a3ce8af64e3e11e4e0ebc09430af9c15e.png, то, очевидно, эти позиции рядом никогда не будут. Отсюда можно сделать вывод, что для фиксированного *l* возможных положений для *r* может быть не больше, чем *K*. Значит, всего таких пар *O*(*NK*). Определим, какой вид должно иметь множество *M*, чтобы после его удаления эти позиции оказались рядом? Нетрудно заметить, что для этого должны выполняться такие 2 условия:

1. http://codeforces.ru/renderer/07de7c5bc68bc92d118ada183dab5d78f5c631c4.png

2. http://codeforces.ru/renderer/7fa1a37ebb11a92726d78e30e0ad17ffda729919.png

Переберем все допустимые пары *l* и *r*, для каждой найдем *M*'. Попутно для каждой тройки (a,b,P) будем хранить сумму стоимостей соседства всех пар *l* и *r*, для которых *Sl* = *a*, *Sr* = *b*, *M*' = *P*. После этого для фиксированного *M* переберем все пары символов (*a*, *b*), а так же подмножества *P* и просуммируем стоимости, которые мы сохранили. Прибавим к этому стоимость выбрасывания символов из *M* и получим конечную стоимость строки. Сложность решения *O*(3*K* \* *K*2 + *NK*).

Попробуем улучшить предыдущее решение, засчет уменьшения множителя при 3*K*. Рассмотрим такой (неправильный) алгоритм:

Переберем все допустимые пары *l* и *r* и для каждой найдем *M*'. Заведем массив *v*, в котором для каждой маски *P* будем хранить сумму стоимостей соседства для всех пар *l* и *r*, для которых *M*' = *P*. Для фиксированного *M* найдем сумму значений из *v* по всем подмаскам *M*, прибавим к ней стоимость удаления *M* и получим конечную стоимость строки.

Этот алгоритм не является правильным, так как некоторые стоимости соседства мы прибавим к суммарной стоимости, но на самом деле позиции не будут соседями, т.к. один из концов (или оба сразу) будут принадлежать *M*, т.е. будут удалены. Воспользуемся формулой включения-исключения и при заполнении массива *v* сделаем следующие действия:

*v*[*M*'] +  = *cost*

http://codeforces.ru/renderer/b4d392e8baa2755f671b67077d5ad638a2818e1c.png

http://codeforces.ru/renderer/0939a5ad00766ddcedb4465081c7dc8c63f0dd22.png

http://codeforces.ru/renderer/872ba326de38444a17ace78f749a43a3e1eff1d1.png

При таком заполнении *v* описанный выше алгоритм будет работать правильно и сложность станет *O*(3*K* + *NK*).

Этого все еще недостаточно, но мы уже почти у цели. Осталось научиться быстро считать сумму в массиве *v* по всем подмаскам *M*. Для этого рассмотрим следующий итеративный алгоритм:

Пусть перед первой итерацией у нас есть массив *v*, в котором хранятся начальные значения. После итерации с номером *i* в *v*[*mask*] будет храниться сумма значений из изначального массива *v* по всем подмаскам *mask*, для которых первые *K* - *i* бит совпадают с соответствующими битами маски *mask*. На итерации с номером *i* для всех масок, в которых *i*-й бит (нумерация с единицы) единичный сделаем сделаем такое: *v*[*mask*] +  = *v*[*mask*\{*i*}]. Нетрудно заметить, что после выполнения *K* итераций этого алгоритма в новом массиве *v* как раз и будут находиться суммы по всем подмаскам из изначального массива *v*.

Сложность алгоритма *O*(2*KK* + *NK*).

**Задача D: Плюс и XOR (Даниил Нейтер)**

Рассмотрим какой-то единичный бит в X. Если соответствующий бит в Y равен 0, то мы можем их поменять местами, уменьшив X и увеличив Y. При этом их сумма и xor не изменятся. Отсюда можно сделать вывод, что если какой-то бит равен единице в *X* то он будет равен единице и в *Y*. Отсюда *Y* = *X* + *B*. Учитывая, что *X* + *Y* = *X* + *X* + *B* = *A*,  получаем следующие формулы для нахождения *X* и *Y*:

*X* = (*A* - *B*) / 2

*Y* = *X* + *B*

Следует также учесть, что если выполняется хотя бы один из следующих пунктов:

* *A* < *B*
* *A* и *B* имеют разную четность
* *X* *and* (*A* - *X*) ≠ *X*, где *and* - побитовое "и"

то ответа не существует и следует вывести -1.

**Задача E: Точки (Даниил Нейтер)**

Если перегруппировать слагаемые, то можно разбить сумму на две:

http://codeforces.ru/renderer/6e7ca6f50695e36ea199c3704ebb7574c4bfc029.png

http://codeforces.ru/renderer/b4a6a53d2cb6e9458620cc0b964c69f610df6601.png

Рассмотрим подсчет суммы по иксам:

http://codeforces.ru/renderer/ce7d437559ce56c7602146506aaec7f5b8f5920b.png

http://codeforces.ru/renderer/14de3bd836c0e823801f884a115983ff447988b5.png

Полученная формула позволяет посчитать ответ за 1 проход. Сложность решения *O*(*N*).

**Задача F: Турист (Илья Порублев)**

Из события *i* можно попасть в событие *j*, если выполняются условия:

* *ti* ≤ *tj*
* |*xi* - *xj*| ≤ |*ti* - *tj*|· *V*

Если события представлять в виде точек на координатной плоскости с координатами (*xi*, *ti*), то предыдущие 2 условия можно представить графически, а именно:

Из события *i* можно попасть в событие *j*, если точка (*xj*, *tj*) лежит внутри угла направленного вверх с вершиной в (*xi*, *ti*), а стороны которого образуют угол *arctg*(*V*) с вертикалью. Сделаем преобразование координат, при котором точка с координатами (*xi*, *ti*) переходит в точку (*pi*, *qi*), где

*pi* =  - *xi* + *ti* \* *V*

*qi* = *xi* + *ti* \* *V*

Тогда условие того что из события *i* можно попасть в событие *j* принимает вид: *pi* ≤ *pj* и *qi* ≤ *qj*. Отсортируем все точки по возрастанию координаты *p*, а при равном *p* - по возрастанию *q*. Задача (та часть, где можно самому выбирать стартовую точку) сводится к тому, чтобы в получившемся массиве найти самую длинную возрастающую подпоследовательность по *q*. Это можно сделать за *O*(*NlogN*).

Чтобы решить часть, в которой турист стартует из точки (0, 0) можно создать фиктивное событие со временем 0 и абсциссой 0 (если такого еще небыло) и потребовать чтобы мы обязательно посетили его первым.

Суммарная сложность решения *O*(*NlogN*).